

# Eines Informàtiques per a l'Estadística: R

Aureli Alabert

Grau d'Estadística Aplicada  
Universitat Autònoma de Barcelona

3 d'octubre de 2023



# 1 Instal·lació del R i el RStudio

## 1.1 Instal·lació del R

El **R** és un programari per a la manipulació, anàlisi i representació gràfica de dades. És gratuït, *open source*, i extensible a partir de paquets (*packages*) que es poden instal·lar addicionalment al sistema bàsic en el moment que es necessitin, i als quals tothom pot contribuir quan es tenen els coneixements suficients.

Per instal·lar-lo, en Windows o MacOS, aneu a

<https://www.r-project.org>

(per Linux vegeu més avall; el vostre gestor de paquets serà suficient).

Seguiu l'enllaç de la segona línia [download R](#), que us porta a una llista de servidors rèplica (*mirrors*). Escolliu un que estigui geogràficament proper (per exemple, <https://cran.rediris.es/>, o bé <https://stat.ethz.ch/CRAN/>), i us portarà a una pàgina on veureu tres enllaços, per diversos Linux, per MacOS, i per Windows. Escolliu el corresponent al vostre sistema operatiu.

### 1.1.1 Linux

En principi, és suficient instal·lar el paquet `r-base` que està en totes les distribucions, mitjançant:

```
sudo apt install r-base
```

Si no funciona, seguiu les instruccions de l'enllaç anterior.

### 1.1.2 MacOS

Descarregueu el fitxer `.pkg` adequat d'aquesta pàgina:

- Si el vostre processador és un Apple M1 o M2, baixeu el `R-4.3.1-arm64.pkg` (el número concret de versió pot haver canviat).
- Si és un Intel amb sistema macOS 11 (Big Sur) o posterior, baixeu el fitxer `R-4.3.1-x86_64.pkg`.
- Si és Intel, però teniu un sistema anterior al Big Sur, busqueu més avall a la pàgina la versió més recent que pugui funcionar en el vostre sistema.

Si dubteu quina versió de sistema o processador teniu, ho podeu veure en el menú **Apple > About this Mac**.

Obriu el `.pkg` que heu baixat i seguiu les instruccions de l'instal·lador. Bàsicament, anireu confirmant tot el que es demana. Al final us preguntarà si voleu enviar l'instal·lador a la paperera; podeu dir-li que sí.

Al acabar, tindreu un fitxer `R.app` en el directori **Applications** i en el **Launchpad**.

### 1.1.3 Windows

Seguiu l'enllaç [base](#) i després clickeu [Download R 4.3.1 for Windows](#), que us baixarà el fitxer `R-4.3.1-win.exe` (el número concret de versió pot haver canviat.) Executeu aquest fitxer.

Us demanarà l'idioma d'instal·lació. Escolliu-lo i després accepteu totes les opcions que va preguntant tal qual s'ofereixen.

## 1.2 Instal·lació del RStudio

Si arrenqueu el programa **R** des dels menús del sistema o launchpad, veureu una finestra amb títol **R Console**. Amb això podeu comprovar que el R està ben instal·lat. Però treballar en aquesta consola és molt incòmode. Tothom utilitza un editor sofisticat, anomenat **RStudio**, que proporciona un entorn complet de desenvolupament (“Integrated Development Environment”, IDE).

Independentment del vostre sistema operatiu, aneu a [posit.co](#), premeu el botó de la dreta [Download RStudio](#) i seguiu els enllaços que van dient [Download RStudio](#) i [Download RStudio Desktop](#). Segurament detectarà el vostre sistema automàticament, i si no, teniu totes les versions en aquesta pàgina.

Específicament per a MacOS: Observeu que la versió més recent requereix macOS 11 (Big Sur) o posterior. Baixeu el `RStudio-....dmg` i obriu-lo. Moveu la icona `RStudio.app` sobre la carpeta `Applications`.

## 2 Engegar el RStudio

Al engegar el **RStudio**, la finestra es veu dividida en diversos panels, que es poden redimensionar arrossegant les vores. De moment, ens concentrem en el panel que té una pestanya que diu **Console**. Hi veiem un missatge de benvinguda i un *prompt* (el símbol `>`), que ens indica que el sistema està preparat per rebre instruccions.

Les instruccions s'acaben amb la tecla **RETURN** (**ENTER**, **INTRO**), i s'executen immediatament. Per exemple, proveu d'escriure `3+2`. La resposta normalment apareix directament sota la instrucció. Si la resposta és un gràfic, sortirà en un altre panel, sota la pestanya **Plots**. El **R** és una eina molt potent per fer gràfics a partir de dades numèriques.

## 3 Primeres passes

**1 §** Hem vist que el **R** sap fer algun càlcul senzill:

```
1 3+2
2 [1] 5
```

També sap fer càlculs una mica més complicats, com ara  $e^{-2}$ :

```
1 exp(-2)
2 [1] 0.1353353
```

Totes les funcions matemàtiques habituals són accessibles escrivint la instrucció oportuna. Per exemple, sinus, cosinus, tangent, logaritme neperià i logaritme en base 10 es calculen d'aquesta manera (proveu-ho amb aquests i altres números):

```
1 sin(0)
2 cos(0)
3 tan(0)
4 log(100)
5 log10(100)
```

Observacions:

- En matemàtiques 'log' fa referència al logaritme neperià, sempre! El logaritme en base 10 es denota ' $\log_{10}$ ', i en qualsevol altre base  $b$ , s'escriu ' $\log_b$ '. És possible que ho hàgiu estudiat abans d'una altra manera; cal que feu el canvi mental. El logaritme neperià també s'escriu de vegades  $\ln$ , o  $\text{Ln}$ .
- Els arguments de les funcions trigonomètriques venen donats en radians, no en graus. Aquest també és el conveni en general en matemàtiques, si no es diu el contrari. Així, podeu comprovar que `sin(90)` no dona 1, com seria el cas si es tractés de graus.

**2 §** Fins aquí una calculadora de butxaca, un *smartphone* o una *tablet* ho poden fer igual de bé. Compliquem-ho una mica utilitzant *variables*, és a dir, noms que poden contenir valors numèrics concrets:

```
1 x <- 3
2 x + 2
3 [1] 5
4 x * (x+2)
5 [1] 15
```

La primera d'aquestes instruccions és una *assignació*. A partir d'aquest moment la variable `x` tindrà el valor `3`. La "fletxa" s'ha d'escriure amb els dos caràcters `<` - seguits; els espais blancs abans i després de la fletxa són opcionals, però queda més llegible si es posen. Encara que l'ordinador ho entengui igual, les instruccions també les hem de llegir els humans, i es bo s'entenguin fàcilment.

La segona instrucció suma 2 unitats al valor contingut a la variable. El **R** ens contesta amb el resultat. Anàlogament amb la tercera instrucció.

Observeu que les *assignacions* no produeixen cap resposta. Per saber el valor que té una variable, només cal escriure una instrucció amb el seu nom. Aquestes dues instruccions produeixen el mateix resultat:

```
1 x
2 print(x)
```

Òbviament, la primera és més còmode, però a partir d'un cert moment veureu que cal usar la segona per a què les coses funcionin.

En la majoria de llenguatges de programació, les assignacions es fan amb el símbol d'igualtat `=` en lloc de la fletxa. El **R** també admet aquest símbol en la major part de les situacions, però no sempre. Per tant, no és aconsellable usar-lo i els experts ho considerem de mal estil.

També és correcte la “fletxa a l'inrevés”. Per exemple `7 -> y` assigna el valor 7 a la variable `y`. Però queda difícil de llegir i per això tampoc no és aconsellable.

**3 §** A més d'assignar un valor numèric directe a una variable, podem assignar-li el resultat d'una expressió involucrant altres variables definides abans:

```
1 x <- 1.4
2 y <- x^2 + 5
3 z <- sqrt(x+y)
```

Observem que:

- S'utilitza un “punt decimal”, no pas una coma.
- El símbol `^` serveix per *exponenciar*: elevar una base a una potència. En aquest cas, estem elevant `x` al quadrat.
- L'arrel quadrada s'obté mitjançant `sqrt()`.

Feu que el **R** escrigui el valor d'aquestes tres variables ara. Observeu que la variable `x`, que més amunt tenia el valor 3, ara té el nou valor 1.4; és a dir, una nova assignació anul·la l'assignació anterior.

Veiem que passa quan usem una variable a la qual no s'ha assignat abans un valor:

```
1 x <- h
```

Heu d'obtenir com a resultat el missatge: `Error: object 'h' not found`. Si imprimeu després el valor de `x` veureu que no ha canviat. La instrucció és invàlida i no canvia res.

Una de les coses més difícils en **R** i en molts altres entorns de computació és entendre els missatges d'error. El missatge anterior és força clar, però de vegades no ho són tant. Prepareu-vos per un cert grau de frustració quan el **R** no us admeti una instrucció; ens passa a tots. Internet pot ajudar molt: proveu de posar a Google o qualsevol altre buscador `R object not found` i veureu la gran quantitat de pàgines que mencionen aquesta frase fent referència al **R**.

**4 §** Una variable no té perquè ser una lletra. Una variable pot ser qualsevol tira de caràcters amb les condicions següents:

- És una combinació de lletres, dígitos, punts (`.`) i blancs subratllats (`_`).<sup>1</sup>
- Comença amb una lletra o un punt. Si comença amb un punt, el següent caràcter no pot ser un dígit.

Per tant, són vàlids noms com ara `B42`, `.a.b`, `0_0`. Però naturalment és molt aconsellable que els noms de les variables tinguin un sentit en el context en que s'utilitzen: És millor `temperatura` i `Velocitat` que `t`, i `V`, per exemple.

Cal tenir en compte que majúscules i minúscules són lletres diferents. Comproveu-ho. Vegeu també quin és el missatge d'error quan intentem assignar un nom de variable invàlid.

<sup>1</sup>Anomenat també “barra baixa”; en anglès, *underscore*.

Hi ha unes quantes *paraules reservades* del **R** que no és permès usar com a variables. Aquestes paraules són, bàsicament,

```
if else repeat while function for in next break  
TRUE FALSE NULL Inf NaN NA NA_integer_ NA_real_ NA_complex_ NA_character_
```

Els que parlem idiomes llatins tenim una tendència natural a usar accents en els noms de variable. Segurament el **R** us admetrà noms com ara *desviació* o *alçada*. Però, per raons llargues d'explicar, és fortament aconsellable restringir-nos a usar els caràcters de l'alfabet anglès. El consell val també per a noms de fitxers, on tampoc haurien d'haver-hi mai espais blancs. A aquests efectes, *ñ* i *ç* també són lletres accentuades.

**5 §** Hi ha alguns noms de variable que ja tenen un valor quan engeguem el **R**, tot i que es poden canviar si es vol. Per exemple, *pi*. Mireu quin valor té, i calculeu el sinus i el cosinus de *pi*. Més endavant veurem com podem ajustar la quantitat de decimals que ens dona el **R** en els resultats.

El sinus de  $\pi$  radians (180 graus) és ben conegut que dona zero. Segurament el **R** no us ha donat zero exactament, sinó una cosa com ara `1.224606e-16`. Això vol dir  $1.224606 \times 10^{-16}$ , o sigui 0.0000000000000001224606, que a tots els efectes pràctics es pot considerar igual a zero.

És inevitable que hi hagi imprecisió en els càlculs matemàtics. Hi ha maneres de reduir aquesta imprecisió tant com es vulgui, però de moment no ens en preocuparem gaire.

**6 §** Quan tenim el focus dins la pestanya **Console** (o sigui, hi veiem el cursor intermitent), podem usar les tecles amunt i avall per recuperar instruccions anteriors i tornar-les a executar. També es pot copiar i enganxar des de la consola o cap a ella, amb les tecles habituals **CTRL-C**, **CTRL-V**.<sup>2</sup>

A la pestanya **History**, que està en un altre panel, apareixen les instruccions que hem usat fins al moment. Si feu doble-click en una d'elles, o premeu el botó **To Console**, es copiarà a la consola i la podeu executar o editar.

Ocasionalment potser voldreu esborrar el que hi ha a la consola. En el menú **Edit** trobareu l'entrada **Clear Console**, que està assignada també a la combinació **CTRL-L**. També ho podeu fer amb la petita icona de l'escombra, a dalt a la dreta del panel. Netejar la consola no esborra la història.

**7 §** L'aspecte visual del **RStudio**, i altres paràmetres, es poden canviar usant **Tools > Global Options...** En particular, a l'apartat **Appearance** es pot canviar el "tema" general i el de l'editor, a més del tipus i mida de lletra.

A l'apartat **Pane Layout** es pot canviar la posició dels panels i les pestanyes de cadascun. Recomanable no tocar-ho fins que no tingueu més "opinió" sobre com us agrada treballar. Com veureu, hi ha de fet quatre panels, encara que fins ara només n'heu vist tres. El panel de dalt a l'esquerra (Source) és un editor de fitxers que veurem més endavant.

**8 §** Per tancar el programa, podeu escriure a la consola `quit()`, o abreviadament `q()`, o simplement tancar la finestra del **RStudio** de la manera ordinària. En tots els casos ens

<sup>2</sup>Pels usuaris de MacOS, **CMD-C**, **CMD-V**. Cada cop que aparegui **CTRL**, feu la traducció.

preguntarà si volem guardar el *workspace*. La resposta ha de ser sempre **NO**, excepte si teniu raons de pes per fer el contrari.

Guardar el workspace vol dir guardar les variables que tenim definides i altres detalls de la sessió en un fitxer que s'anomena *.RData*. Aquest fitxer es llegeix la propera vegada que engeguem el **R**, i ens deixa en l'estat en què vam tancar. És mala idea guardar el workspace, perquè pot interferir amb la nova sessió que estem obrint. Veurem que hi ha altres maneres molt més segures de guardar informació per reutilitzar-la fiablement.

Quan us canseu de contestar sempre **NO!**, podeu obrir les **Global Options...** i a l'apartat **General** desmarcar la casella "Restore *.RData* ..." i posar Never en les opcions de "Save workspace to *.RData* ...".

## 4 Help!

**9 §** La manera bàsica d'obtenir ajuda és usar la funció `help()` a la consola. Com a argument es posa el nom d'allò sobre el que volem ajuda, i l'explicació apareix a la pestanya **Help**. Per exemple, proveu

```
1 help(sin)
2 help(log10)
3 help(sqrt)
```

L'explicació conté una descripció, la sintaxi, els arguments (allò que ha d'anar dins els parèntesis), el valor que retorna, exemples d'ús, i potser unes quantes funcions relacionades.

Quan s'està mostrant una ajuda, podeu fer click sobre la icona del panel amb tooltip **Show in new window**. L'ajuda s'obrirà en una finestra apart del **RStudio**.

Si voleu tenir l'ajuda guardada en un fitxer, potser per imprimir-la o llegir-la en una altre context, podeu fer `help(sqrt, help_type="pdf")`. El fitxer pdf es guarda, en principi, en el nostre directori *home*.<sup>3</sup>

En el menú principal del **RStudio** tenim l'entrada **Help > R Help**, des d'on es pot navegar, a la pestanya **Help**, a tota mena de documentació sobre **R** i **RStudio**. Per exemple, si seguïu els enllaços a **Packages** i després **base** veureu la llista de totes les funcions contingudes en el "paquet" anomenat *base*, que és un dels que estan "carregats en el R" automàticament quan l'engeguem. Més endavant veurem què vol dir un *package*.

**10 §** La informació que hi ha a internet sobre el **R** és immensa. Quan estiguen aturats amb una cosa que no us funciona, useu Google o qualsevol buscador amb la instrucció que no va o el missatge d'error que no enteneu. La probabilitat és molt alta que en poca estona estiguen altre cop en marxa.

Al paràgraf 4 § vam veure la llista de paraules reservades que no es poden usar com a noms de variables en **R**. Aquesta llista també es pot veure fent `help(reserved)`. Proveu-ho. Veureu que a més d'aquelles paraules, tampoc es poden usar variables que comencin per punt-punt-dígit, o per punt-punt-punt.

<sup>3</sup>El típic **Documents** en Windows.



## 5 Objectes

**11 §** Fins ara hem vist variables i com assignar valors numèrics a una variable, amb instruccions com ara `x <- 3`. Veurem ara que el **R** manipula en general *objectes*, i el que hem anomenat fins ara *variables* és només un dels tipus d'objectes que hi ha.

Tot objecte té un tipus o *mode*. Els modes bàsics són: “numeric”, “character”, “logical”. Proveu:

```
1 x <- 3
2 mode(x)
3 x <- "pastanaga"
4 mode(x)
5 x <- TRUE
6 mode(x)
```

El mode és *numèric* quan la variable conté un nombre, tant sigui enter com amb decimals. El mode és *character* quan la variable conté una cadena de caràcters. Les cadenes estan sempre delimitades per cometes. El mode és *logical* quan la variable conté algun dels valors `TRUE` o `FALSE`.

Observeu que al canviar el valor assignat a una variable podem canviar el seu mode. Això pot ser convenient en algun moment, però també té els seus perills, en forma d'errors difícils de trobar.

**12 §** Un *vector* és un objecte que conté una col·lecció de valors del mateix mode. Els vectors es creen amb la funció `c()` (penseu-la com abreviatura de “concatenar”), i s'assignen a un nom de la mateixa manera que ja hem vist. Per exemple:

```
1 weight <- c(60, 72, 57, 90, 95, 72)
2 weight
3 [1] 60 72 57 90 95 72
4 mode(weight)
```

Hem creat un vector numèric que conté elements. Quan invoquem el nom del objecte, ens imprimeix els sis valors. Si voleu saber què vol dir el [1] que apareix abans de cada sortida, construïu un vector llarg, de manera que al imprimir-lo el resultat ocupi més d'una línia.

Una variable és a tots els efectes un vector que té un sol element.

Suposem que els valors que hem dipositat a la variable `weight` són els pesos, en kg, de sis persones, i que les seves alçades, en metres, i donades en el mateix ordre, són les que posarem ara en el vector `height`:

```
1 height <- c(1.75, 1.80, 1.65, 1.90, 1.74, 1.91)
2 bmi <- weight/height^2
3 bmi
4 [1] 19.59184 22.22222 20.93664 24.93075 31.37799 19.73630
```

A la segona línia estem dividint el pes pel quadrat de l'alçada, obtenint un valor conegut com a “body mass index”. Observeu que la variable `bmi` on hem dipositat el resultat

conté també un vector. La fórmula s'ha aplicat component a component als dos vectors involucrats. Aquesta és una característica fonamental del **R**:

*Les operacions entre vectors es realitzen component a component.*

Una altra característica relacionada és la anomenada “regla de reutilització”:

*Si, al fer operacions entre vectors, un d'ells té una longitud més curta que els altres, quan arribem al final es torna a llegir des del principi.*

Per exemple, `height*c(1,-1)` produeix un vector on els components parells de `height` queden amb el signe canviat. Comproveu-ho. Aquesta característica també l'hem usada a la fórmula de la segona línia: El 2 que representa elevar al quadrat es considera com un vector d'un sol component, que anem reutilitzant fins que s'acaba el vector `height`.

Observeu que en l'expressió `height*c(1,-1)` hem usat un vector sense donar-li nom prèviament. I com que tampoc no hem assignat el resultat a un nom, el **R** ens l'imprimeix directament.

De vegades convé crear vectors buits. Ho fem amb les funcions `numeric()`, `character()`, `logical()`.

```
1 x <- numeric()
2 x
3 mode(x)
4 x <- character()
5 x
6 mode(x)
7 x <- logical()
8 x
9 mode(x)
```

També podem crear vectors de longitud determinada sense especificar el seu contingut. La longitud es passa com a argument a les funcions anteriors. El resultat depèn del mode del vector. Observeu el que surt:

```
1 x <- numeric(10)
2 x
3 x <- character(10)
4 x
5 x <- logical(10)
6 x
```

Com veieu, hi ha un contingut “per omissió”, que en el cas del mode caràcter, és un vector de deu cadenes de caràcters buides, que s'imprimeixen amb cometes sense res en mig.

**13 §** Una *llista* és un objecte que conté una col·lecció de valors, no necessàriament del mateix mode. Creem una llista amb la funció `list()`. Proveu:<sup>4</sup>

```
1 dades <- list("Red", c("Orange", "Green"), 42,
2             c(3, 14, 16), TRUE, c(FALSE, FALSE))
```

<sup>4</sup>Podeu escriure-ho en una sola línia, o bé en més d'una. En aquest últim cas, veureu que el *prompt* canvia a `+`, indicant que no hem acabat la instrucció.

Imprimiu l'objecte creat i observeu molt bé com és la sortida i assegureu-vos que l'enteneu. Podeu provar de construir altres llistes. En particular observeu que dins una llista poden haver-hi vectors, i de fet, altres llistes:

```
1 dades <- list(list("Red", 0, FALSE), list("Green", 1, TRUE))
```

Imprimiu i observeu atentament la sortida.

**Exercici:** Feu una llista en què cada element contingui el nom i l'edat dels membres de la teva família.

**14 §** Un cop construïts un vector o una llista, hem de poder accedir als seus elements individuals. En el cas dels vectors, escriurem, per exemple, `weight[3]` per accedir al tercer element. En el cas de les llistes, usarem dos parells de parèntesis quadrats: `dades[[2]]`.

**Exercici:** Com que aquest `dades[[2]]` és una altra llista, hem de poder accedir als seus elements individuals. Com creieu que es fa? Quina diferència hi ha entre aquestes dues instruccions?:

- `dades[[2]][1]`
- `dades[[2]][[1]]`

**15 §** Trobareu que molt sovint heu de crear vectors amb seqüències sistemàtiques de nombres. Això es pot fer amb la funció `seq()` (abreviatura de “sequence”). En la forma més bàsica, la funció pren tres arguments, `from`, `to`, `by`. Proveu com funciona amb els exemples següents:

```
1 y1 <- seq(from=1, to=100, by=1)
2 y2 <- seq(from=1, to=100, by=2)
3 y3 <- seq(from=50, to=100, by=10)
4 y4 <- seq(from=100, to=1, by=-1)
5 y5 <- seq(from=-5, to=5, by=0.2)
```

Si posem, per exemple, `y2 <- seq(1, 100, 2)` funciona igual (comproveu). Les paraules `from`, `to`, `by` són els *noms dels arguments* de la funció `seq`. Podem posar-los o no. Si no els posem, hem de posar els arguments exactament en l'ordre anterior. En canvi, usant els noms dels arguments, els podem posar en qualsevol ordre (comproveu). A més, amb els noms posats, és segurament més fàcil d'entendre el que fa la funció. Ara no té molta importància, però ja veureu que escriure instruccions clares la tindrà. I és més fàcil recordar els noms dels arguments que l'ordre en què s'han d'especificar.

L'expressió `1:15` és una manera abreviada de fer `seq(from=1, to=15, by=1)`. Proveu:

```
1 1:15
2 2*1:15
3 2*(1:15)
4 (2*1):15
5 15:1
```

Observeu que els dos punts (`:`), tenen *precedència* respecte la multiplicació. Per desfer la precedència, cal usar parèntesis. En cas de dubte, millor sempre usar parèntesis. A més, l'expressió `2*(1:15)` és més clara que `2*1:15`.

La funció `rep()` (de “repeat”) s'utilitza per construir un vector repetint un objecte una quantitat especificada de vegades.

```
1 rep(8, times=5)
2 x <- 1:4
3 rep(x, times=5)
```

Aplicat a llistes, `rep()` “concatena” les llistes en una nova llista. No fa una “llista de llistes”:

```
1 salutacions <- list("hola", "adéu")
2 rep(salutacions, times=3)
```

La funció `length()` retorna la longitud d'un vector o llista.

```
1 length(y5)
2 length(salutacions)
3 length(rep(salutacions, times=3))
```

La longitud, igual que el mode, és un atribut que tenen tots els objectes del **R**.

**16 §** Hem vist abans que hi ha objectes de mode *logical*. Anem a treballar amb ells. Recordeu les definicions dels vectors `weight`, `height`, `bmi`, del paràgraf 12§. Comproveu que encara els teniu definits o torneu-los a entrar.

Es diu que una persona té sobrepès si el seu índex de massa corporal (el `bmi`) és superior a 22.5. L'expressió `bmi > 22.5` té un valor `TRUE` o `FALSE` segons l'expressió sigui veritat o mentida. Com que `bmi` és un vector, el resultat serà un vector *logical*, que ens dirà si cadascuna de les persones de la llista té sobrepès o no.

```
1 sobrepes <- bmi > 22.5
2 bmi[sobrepes]
```

Què ha fet la segona instrucció? Dins dels parèntesis quadrats hem posat un vector lògic. El resultat és seleccionar només aquells components en què el vector lògic té el valor `TRUE`. Queda un vector de longitud més petita.

Hi ha altres maneres de seleccionar només una part d'un vector. Per exemple, proveu i mireu d'entendre què fan:

```
1 bmi[1:4]
2 bmi[2:5]
3 bmi[c(1,1,1,6,6,6)]
4 bmi[-6]
5 bmi[-(1:3)]
```

De la mateixa manera que seleccionem una part d'un vector per construir un altre vector, també podem canviar els valors del vector original. Experimenteu:

```
1 x <- 1:20
2 x[3] <- 10
3 x[7:9] <- 0
```

i tot el que se us pugui acudir.

El símbol `>` és un exemple de *operador lògic*. Representa la pregunta “és el que hi ha a l'esquerra més gran que el que hi ha a la dreta?”, i la resposta només pot ser, naturalment, `TRUE` o `FALSE`.

```
1 4 > 3
2 3 > 4
3 3 > 3
4 x <- 1:5
5 x > 3
6 y <- 3:7
7 x > y
```

Reflexioneu sobre l'últim resultat fins que el tingueu clar. Recordeu que les operacions entre vectors es fan “component a component”.

Experimentem més:

```
1 x <- 1:5
2 length(x)
3 y <- -5:14
4 length(y)
5 x > y
```

Reflexioneu sobre l'últim resultat recordant la “regla de reutilització”.

Fem encara un altre experiment:

```
1 x <- 1:5
2 length(x)
3 y <- -5:12
4 length(y)
5 x > y
```

Obtenim un resultat, però també un “warning”. El **R** no considera que sigui un error, però ens avisa que el que estem fent és estrany i potser no és el que volíem. El vector `y` té una longitud que no és múltiple de la longitud de `x`. Per tant, quan anem reutilitzant `x` arriba un moment que el vector `y` s'acaba i en canvi el `x` està a mitges. El resultat és un vector més curt. Les dues últimes comparacions no s'han fet.

Hi ha altres operadors lògics:

`<`   `<=`   `>`   `>=`   `==`   `!=`

que pregunten, respectivament,

- és més petit que?
- és més petit o igual que?
- és més gran que?
- és més gran o igual que?
- és igual que?
- és diferent que?

Els dos últims són els que tenen una sintaxi menys intuïtiva, i és molt fàcil equivocar-se amb ells. Si fem `x = 3` no estem preguntant. Això és una assignació, equivalent a `x <-`

3 com vam comentar al paràgraf 2 §. Acostumeu-vos a no usar mai el símbol = sol, i reduireu la probabilitat d'errors.

**Exercici:** Experimenteu amb tots ells per comprovar que els enteneu.

Hi ha encara uns altres operadors lògics que operen entre valors o vectors lògics. Si  $p$  i  $q$  són vectors lògics,  $p \& q$  és la seva *conjunció*,  $p | q$  és la seva *disjunció*, i  $!p$  és la negació de  $p$ .

- La conjunció de dos valors lògics és **TRUE** si i només si tots dos valors són **TRUE**.
- La disjunció és **TRUE** si i només si almenys un dels dos valors és **TRUE**.
- La negació és **TRUE** si i només si el valor original és **FALSE**.

**Exercici:** Intenteu endevinar els valors que sortiran en les tres últimes instruccions, abans d'executar-les:

```
1 p <- c(TRUE, TRUE, FALSE, FALSE)
2 q <- c(TRUE, FALSE, TRUE, FALSE)
3 !p | q
4 !(p & !q) | p
5 (p & !q) & !p
```

**17 §** Un vector numèric o lògic pot contenir també el valor **NA**, que es llegeix com “Not Available”. És un valor indefinit, i normalment una operació que involucri un **NA** donarà com a resultat un **NA**. Per exemple:

```
1 x <- 1:5
2 x <- c(x, NA)
3 x > 3
4 is.na(x)
```

A la segona línia hem afegit al final del vector  $x$  un valor **NA**. Veiem com el resultat de la comparació és també **NA** per aquest valor. A l'última línia hem usat una funció que retorna **TRUE** o **FALSE** segons si el l'argument és **NA** o no.

Els valors **NA** es poden produir com a resultat de certes manipulacions. Per exemple, si tenim un vector de 5 components, com el  $x$  anterior, i fem  $x[6] <- 0$  estem afegint un sisé element (és equivalent a  $x <- c(x, 6)$ ). Si fem  $x[10] <- 0$ , ens afegirà uns quants valors **NA** per completar els deu elements que ara tindrà el vector. Comproveu totes aquestes afirmacions.

Hi ha un altre valor especial que es pot produir com a resultat d'un càlcul: El valor **NaN**, que es llegeix com “Not a Number”. Per exemple, intenteu fer l'arrel quadrada d'un número negatiu.

També produeixen **NaN**'s les operacions  $0/0$  i  $Inf - Inf$ . En canvi no ho fan les operacions  $1/0$ ,  $-1/0$ ,  $Inf-5$ ,  $2*Inf$ ,  $(-1)*Inf$ . Comproveu-ho. **Inf** representa un valor més gran que qualsevol nombre real, i que habitualment anomenem *infinit*. (Si heu estudiat *límits*, veureu que els resultats de les operacions anteriors coincideixen amb les idees que ja teniu sobre límits.)

La funció `is.na()` que hem vist abans retorna **TRUE** tant si l'argument és un **NA** com un **NaN**. Per distingir-los hi ha la funció `is.nan()`, que només retorna **TRUE** per **NaN**'s.

**Exercici:** Intenteu predir quin serà el resultat de les següents operacions i comproveu-ho:

```
1 y <- c(3, -2, 5, NA, 4, -1, 7)
2 y[y>0]
3 y[!is.na(y) & y>0]
4 y[!is.na(y) & y<0] <- -y[!is.na(y) & y<0]
5 print(y)
```

**18 §** Si volem saber els noms dels objectes que tenim definits, podem usar la funció `objects()` o, més curt, `ls()`, abreviatura de ‘list’. Proveu-ho ara.

Per “esborrar” un nom d’objecte (o sigui, tornar-lo a deixar indefinit) s’usa la funció `rm()` (penseu-ho com abreviatura de ‘remove’). L’argument ha de ser un nom de variable o una llista de noms de variable. Feu per exemple

```
1 rm(x)
2 x
```

Quan tanqueu la sessió de **R**, recordeu de no guardar la “workspace image”. Així tindreu una sessió neta, sense objectes, la propera vegada que l’engegueu.

## 6 Scripts

**19 §** El **R** permet treballar interactivament com hem fet fins ara, però també permet executar *scripts*, que són programes fets d’instruccions de **R**. Els scripts es guarden en fitxers, i es poden executar sempre que es vulgui. És la manera de guardar feina feta per poder-la reutilitzar, millorar, etc, com qualsevol altre document.

Des del menú principal del **RStudio**, feu **File > New File > R Script**.

S’obrirà un quart panel que no havíem vist fins ara: l’editor de fitxers, de moment amb una sola pestanya que posa **Untitled1** i contingut buit.

Escriviu en aquesta finestra les línies següents:

```
1 weight <- c(60, 72, 57, 90, 95, 72)
2 height <- c(1.75, 1.80, 1.65, 1.90, 1.74, 1.91)
3 weight
4 height
5 bmi <- weight/height^2
6 bmi
7 bmi[bmi>22.5]
8 which(bmi>22.5)
```

Bàsicament, són instruccions que hem usat abans, excepte la última: La funció `which()` actua sobre un vector lògic i ens diu quines components del vector tenen valor **TRUE**.

Guardeu el fitxer: Feu **File > Save As ...**, poseu-li de nom `script1.R` i guardeu-lo on us sembli bé.

Ara podem fer diverses coses: Si posem el cursor sobre una línia i premem el botó **Run** del panel, executarem aquesta línia com si l’haguéssim escrit a la consola. De fet, la instrucció queda escrita a la consola, i es guarda a la pestanya **History**. Per exemple,

executeu la primera línia i després la tercera. El contingut de `weight` s'imprimeix a la consola.

Si seleccionem diverses línies i després fem **Run**, s'executen les línies seleccionades. Proveu-ho amb les quatre primeres línies, i comproveu que el resultat és l'esperat.

Però el més interessant és executar tot el fitxer amb el botó **Source** de dalt a la dreta del panel. A la consola apareix la instrucció de **R** que el **RStudio** ha cridat al prémer el botó: `source( ... )`, amb el camí complet al nostre fitxer dins els parèntesis.

Per què no surt cap resultat a la consola? Perquè ens ha faltat usar la funció `print()`, que és imprescindible quan volem executar scripts d'aquesta manera (que en el futur serà la majoria de vegades!)

Afegiu els `print()` explícits que manquen i guardeu el fitxer **File > Save As ...** amb el nom `script2.R`. Premeu **Source** altre cop i ara sí sortiran els resultats.

```
1 weight <- c(60, 72, 57, 90, 95, 72)
2 height <- c(1.75, 1.80, 1.65, 1.90, 1.74, 1.91)
3 print(weight)
4 print(height)
5 bmi <- weight/height^2
6 print(bmi)
7 print(bmi[bmi>22.5])
8 print(which(bmi>22.5))
```

Tanqueu el fitxer, amb la creueta de la pestanya. Comproveu que repetint l'ordre `source( ... )` a la consola, s'executa el script, encara que no estigui obert a l'editor.

**20 §** Torneu a obrir el fitxer `lfilescript2.R`, usant el menú **File > Open File ....** Encara que surten els resultats, no queda clar què és cada cosa. Ens podem ajudar posant més `print()`. Aprofitem per introduir alguna informació més:

```
1 weight <- c(60, 72, 57, 90, 95, 72)
2 height <- c(1.75, 1.80, 1.65, 1.90, 1.74, 1.91)
3 print("nombre de persones:")
4 print(length(weight))
5 print("pesos:")
6 print(weight)
7 print("alçades:")
8 print(height)
9 bmi <- weight/height^2
10 print("index de massa corporal (bmi):")
11 print(bmi)
12 print("quants tenen bmi excessiu:")
13 print(length(bmi[bmi>22.5]))
14 print("valor dels bmi excessius:")
15 print(bmi[bmi>22.5])
16 print("persones amb bmi excessiu:")
17 print(which(bmi>22.5))
```

Guardeu-ho com a `script3.R` i executeu-lo.

La instrucció que hem introduït ara per imprimir la quantitat de persones amb `bmi` per sobre de 22.5, té una alternativa amb truc: La funció `sum()` suma totes les components



d'un vector. Els vectors lògics són com vectors numèrics on `TRUE` és igual a 1, i `FALSE` és igual a zero. Per tant, també podríem escriure

```
1 print(sum(bmi > 22.5))
```

Com ja veureu, si un script és una mica complicat, és fàcil oblidar-se de què fa, o com ho fa. Dins un script podem deixar-hi *comentaris* que ens ajudin a recordar-ho. Si una línia conté el símbol `#`, tot el que hi ha escrit darrera no es llegeix com a instrucció de **R**; per tant podem posar-hi el que vulguem.

Per exemple, un pot posar una primera línia tipus

```
1 # Programa que troba persones amb sobrepès a partir de pes i alçada
```

i potser en algun lloc pel mig

```
1 # Buscar si el 22.5 té alguna base científica
```

**21 §** El **R** manté en tot moment un *working directory*, el directori de treball. Podem saber quin és fent `getwd()` (abreviatura de 'get working directory'). Si fem a la consola, per exemple, `source("script3.R")`, el **R** buscarà aquest fitxer només en el directori de treball; si no hi és, sortirà un error. En tal cas, li hem d'especificar el camí complet al fitxer, per exemple `source("D:/UAB/Eines/R/script3.R")`.

El directori de treball es pot canviar amb `setwd()` (abreviatura de 'set working directory'). Per exemple, `setwd("D:/UAB/Eines/R/")`. Si tenim el fitxer en qüestió ja obert, el més pràctic és mitjançant el menú `Session > Set Working Directory > To Source File Location`.

Fixeu-vos, sobretot els que useu Windows, que les barres inclinades són aquestes (`/`) i no aquestes (`\`).

## 7 Més objectes: matrius i data frames

**22 §** Una *matriu* és un altre tipus d'objecte del **R**. És un conjunt d'elements del mateix mode, com els vectors, però disposats en un rectangle. És un concepte de moltíssima utilitat en Estadística. Una matriu té un cert nombre de files i un cert nombre de columnes. Es crea amb la funció `matrix()`, a partir d'un vector, donant la quantitat de files i de columnes.

```
1 x <- 1:24
2 print(x)
3 m <- matrix(x, nrow = 6, ncol = 4, byrow = FALSE)
4 print(m)
5 m <- matrix(x, nrow = 6, ncol = 4, byrow = TRUE)
6 print(m)
```

Si no s'especifica `byrow`, pren el valor `FALSE`, i per tant la matriu es construeix per columnes. Es diu que `FALSE` és el valor *per omissió* del paràmetre `byrow`. I com que la quantitat

de columnes es dedueix de la quantitat de files, i viceversa, només cal posar una de les dues coses (però millor posar les dues, per claredat i per detectar possibles errors).

Fixeu-vos en les etiquetes de files i columnes que posa el **R** en la sortida. És el que s'ha d'escriure per seleccionar com a vector una fila o columna. Per exemple:

```
1 print(m[2,])
2 print(m[,3])
```

Una matriu té dues “dimensions”. Un *array* és una generalització a més dimensions. En lloc de files i columnes, especifiquem un vector donant la magnitud de cada dimensió:

```
1 x <- 1:24
2 m <- array(x, dim=c(3, 4, 2))
3 print(m)
```

Ara, `m` és un array amb 3 “files”, 4 “columnes” i 2 “capes” (o com vulgueu anomenar-ho). Fixeu-vos com ho presenta el **R** quan li demanem que escrigui `m`.

Les regles de reutilització s'apliquen com sempre. Anem a construir un array 3-dimensional amb les mateixes dimensions que el `m`, contenint el número 2 a tot arreu. Després farem la suma i el producte dels dos arrays, que com veureu són operacions que es fan element a element<sup>5</sup>.

```
1 n <- array(2, dim=c(3,4,2))
2 print(n)
3 m_mes_n <- m+n
4 print(m_mes_n)
5 m_per_n <- m*n
6 print(m_per_n)
```

**23 §** Al 13§ hem parlat de llistes. Els elements d'una llista poden tenir noms, per tal d'accedir-hi sense necessitat de saber quin és l'ordre de la llista, que sol ser irrellevant.

```
1 Simpsons <- list(home="Homer", dona="Marge",
2                 nombre.fillls=3, edat.pares=c(36, 34),
3                 edat.fillls= c(10, 8, 1))
```

Ara, en lloc de posar `Simpsons[[1]]`, podem posar `Simpsons[["home"]]`. I encara hi ha una manera més convenient i més usada: `Simpsons$home`.

Per exemple, intenteu entendre bé cadascuna de les instruccions següents:

```
1 Simpsons
2 Simpsons$dona
3 Simpsons$edat.fillls
4 Simpsons$edat.fillls[2]
5 n_fillls <- "nombre.fillls"
6 Simpsons[[n_fillls]]
7 Simpsons <- c(Simpsons, list(nom.fillls=c("Bart", "Lisa", "Maggie")))
8 Simpsons$nom.fillls
```

<sup>5</sup>Si heu estudiat matrius com a objecte matemàtic, sabreu que aquest producte *no* és el que s'anomena en matemàtiques “producte de matrius”, molt més útil.

Les matrius també poden tenir noms a les files i les columnes: Tornem a la matriu `m` del paràgraf 22 §:

```
1 m <- matrix(1:24, nrow = 6, ncol = 4, byrow = FALSE)
2 colnames(m) <- paste("C", 1:4, sep=" ")
3 rownames(m) <- paste("F", 1:6, sep=" ")
4 m
5 m[ , "C2"]
6 m["F4", ]
```

Per posar noms a les columnes i files hem usat la funció `paste()`, que enganxa diferents objectes. Feu `help(paste)` per veure la idea. Aquí, per exemple, estem enganxant la cadena "C" amb cadascun dels números 1,2,3,4 (estem reutilitzant la "C"), sense deixar espai de separació. El resultat són les cadenes "C1", "C2", etc. La separació per ommissió és un espai (`sep=" "`).

Veiem en les últimes instruccions com podem usar els noms de files i columnes per seleccionar una determinada fila o columna. També és possible seleccionar una submatriu:

```
m[ , c("C2", "C4")]
```

**24 §** Un altre tipus d'objecte de **R** és el *data frame*. Escriviu a la consola

```
1 airquality
```

El **R** ve amb unes quantes dades d'exemple, que es poden cridar a la consola directament pel seu nom. Més endavant parlem d'aquests *datasets*. El `airquality` és un d'ells.

Veureu que us ha sortit un conjunt de números organitzat en files i columnes. Aquesta és la manera habitual com es presenten les dades en Estadística. Típicament, cada fila és un "cas" o "observació" (un individu d'una població), i cada columna representa una "variable" (una característica de cada individu). En aquest cas, els individus són dies concrets en què es van prendre unes mesures de qualitat de l'aire a la ciutat de Nova York, i les variables són: La proporció d'ozó, la radiació solar, el vent, la temperatura, el mes, i el dia del mes.

Aquesta taula és un exemple de data frame. Els data frames es poden pensar com a llistes, els components de les quals són vectors (les columnes del data frame). Per exemple, la instrucció `airquality$Ozone` ens retorna el vector numèric de les mesures d'ozó. Els **NA** (vegeu el 17 §) corresponen segurament a dies en què no es va poder fer la mesura o es van perdre després les dades. En aquests casos es parla en general de *missing data*.

Els data frames també es poden pensar com arrays 2-dimensionals (matrius), i per tant també es pot accedir a les dades com en els arrays. En el tros de codi següent, seleccionem la columna de les temperatures i calculem la seva mitjana amb la funció `mean()`; després fem el mateix però agafant només les temperatures del més d'agost; la última instrucció fa el mateix que la tercera, fent la selecció a l'estil dels arrays.

```
1 temperatures <- airquality$Temp[ !is.na( airquality$Temp)]
2 mean(temperatures)
3 temperatures_agost <- airquality$Temp[airquality$Month == 8]
4 mean(temperatures_agost)
5 airquality[airquality[ , 5] == 8, 4]
```

**Exercici:** Què passa si volem fer així la mitjana de la radiació solar el mes d'agost? Com li traiem els `NA`, seleccionem agost i calculem la mitjana en una sola instrucció?

Mireu les dades anomenades `swiss` i feu `help(swiss)` per veure què són. També estan organitzades en un *data frame*, i la única diferència apreciable amb `airquality` és que els casos (és a dir, les files) tenen noms en lloc d'un número d'ordre. Aquest noms formen un vector de caràcters i s'hi accedeix amb la funció `row.names()`.

```
1 row.names(swiss)
2 swiss[row.names(swiss)=="Courtelary", ]
```

En algunes circumstàncies pot ser còmode veure el *data frame* en una graella. Podem fer-ho amb la funció `View()`. Proveu `View(swiss)`. Es pot fins i tot editar manualment un *data frame* usant `swiss <- edit(swiss)`, que obre una finestra externa del **RStudio** i guarda altre cop els canvis en el dataset `swiss`. Podeu provar-ho; no passarà res greu.<sup>6</sup> De tota manera, *no és gens aconsellable* editar un *data frame* d'aquesta manera, perquè no queda constància de quins canvis s'han fet.

**25 §** Anem a veure com es fan algunes operacions amb *data frames* mitjançant exemples petits. En primer lloc, creem un *data frame* nou amb la funció `data.frame()`:

```
1 dades <- data.frame(x = 0:5, y = "a", z = c(TRUE, FALSE))
2 print(dades)
```

Observeu la reutilització dels vectors que són massa curts.

Podem afegir-hi columnes usant un nom nou, igual com ho faríem amb llistes:

```
1 dades$x2 <- 6:1
2 dades$x3 <- "b"
3 print(dades)
```

Podem seleccionar un tros d'aquest *data frame* usant els índexs de files i columnes com si fos una matriu, normalment amb alguna condició al seleccionar els índexs. Per exemple, si volem un *data frame* que contingui les dades anteriors, però només de les files que tenen la columna `z` igual a `TRUE`, podem fer

```
1 dades[dades$z == TRUE, ]
```

o bé amb la funció `subset()`, amb què no cal escriure tant i s'entén igual:

```
1 subset(dades, z == TRUE)
```

Observeu que no hem modificat el *data frame* amb aquestes dues últimes instruccions. Per modificar-lo hem de fer una assignació `<- subset(dades, z == TRUE)`, al mateix `dades` o a un altre objecte.

Quan dos *data frames* tenen variables (columnes) en comú, podem construir un nou *data frame* ajuntant les columnes de tots dos:

```
1 dades2 <- data.frame(x = 0:5, y2 = 10:15, y3 = c(TRUE, FALSE, TRUE))
2 print(dades2)
3 merge(dades, dades2, by = "x")
```

<sup>6</sup>El proper cop que treballem amb **R** aquest dataset estarà com abans. Ja veureu per què.

## 8 Gràfics i funcions

**26 §** El **R** és molt bo fent gràfics de tota mena. Aquí veurem només com fer la gràfica d'una funció matemàtica  $f(x)$ : La instrucció `plot(x,y)`, on **x** i **y** són dos vectors de la mateixa longitud dibuixa el punts  $(x[1], y[1])$ ,  $(x[2], y[2])$ , etc, en uns eixos de coordenades. Proveu:

```
1 x <- c(1,2,3,4)
2 y <- c(4,3,5,3)
3 plot(x,y)
```

Hi ha molts arguments addicionals per ajustar colors, símbol per als punts, rang dels eixos horitzontal i vertical, afegir text, etc.

Anem a dibuixar la funció  $f(x) = \sqrt{x} + \sin(x)$  entre 0 i 10. Per això, primer creem un vector que contingui el valors del 0 al 10 de centèsima en centèsima; després creem un vector amb les imatges de tots aquests punts; finalment fem el dibuix.

```
1 x <- seq(from=0, to=10, by=0.01)
2 y <- sqrt(x)+sin(x)
3 plot(x, y)
```

Quedarà més bonic si poseu `plot(x, y, type="l")`, que uneix els punts amb una línia fina en lloc de dibuixar els punts en sí. Proveu-ho.

El gràfic apareix a la pestanya **Plots**. Des d'allà podeu seleccionar el botó **Export** per guardar-lo en diferents formats (pdf, png, jpeg, etc).

**27 §** Us haureu fixat que en diem *funcions* de **R** a expressions com ara `getwd()`, `plot(x,y)`, `mean(x)`, on apareixen parèntesis amb (potser) alguns objectes de **R** dins.

El **R** és de fet un llenguatge de programació, com ho són el Python, el C, el Java, etc. Un *programa* és una seqüència d'instruccions donades en un determinat llenguatge, que nosaltres escrivim i entenem, i que després es tradueixen a un llenguatge que les màquines entenen (simplificant molt, podem dir que es tradueixen a "zeros i uns"). En **R** podeu dir indistintament *programa* o *script*.

Tot llenguatge de programació modern implementa el concepte de funció: Una *funció* és un conjunt d'instruccions del llenguatge que s'executen totes al *cridar a la funció*. És útil usar una funció quan aquest conjunt d'instruccions fan una tasca concreta, que volem poder fer repetidament sense haver de reescriure cada cop les instruccions, o que fan que el script que crida la funció quedi més fàcil de llegir.

Per exemple, està clar que és més pràctic usar `mean(x)` i `var(x)` que no pas escriure cada cop la fórmula de la mitjana i la variància, adaptant-la a més cada vegada a si el vector en qüestió és diu **x** o d'una altra manera i a la seva longitud concreta.

El **R** ens proporciona una bona quantitat de funcions predefinides des que s'engega. Algunes codifiquen fórmules matemàtiques senzilles, com la de la mitjana i la variància; d'altres són més complicades, com ara la funció `seq()` per crear un vector; i d'altres requereixen molts coneixements informàtics per crear-les: canviar el directori de treball amb `setwd()`, carregar un script amb `source()`, fer sortir un resultat a la consola amb `print()`, o crear un gràfic amb `plot()`.

El que hi ha dins dels parèntesis quan cridem una funció és la *llista d'arguments*. Poden ser un, com en `mean(x)`, més d'un, com en `plot(x, y, type="l")`, o cap, com en `getwd()`. Fins i tot la mateixa funció pot acceptar una quantitat variable d'arguments, com hem vist al dibuixar la gràfica, i els arguments poden ser de diferents tipus<sup>7</sup>.

Molts cops, els arguments tenen *nom d'argument* i els podem usar dins els parèntesis. Ho hem vist amb la funció `seq()` al paràgraf 15§. Si no posem els noms d'argument, aquests han d'anar en un ordre determinat; si els posem, els podem especificar en qualsevol ordre. Proveu:

```
1 seq(10, 100, 2)
2 seq(from=10, to=100, by=2)
3 seq(by=2, from=10, to=100)
```

A més, els arguments poden tenir *valors per omissió*. Deduïu-los:

```
1 seq(from=1, to=100)
2 seq(to=100, by=2)
3 seq(from=1, by=2)
4 seq(to=100)
5 seq()
```

La funció `seq()` té a més altres arguments alternatius: podem donar la longitud del vector a crear, bé directament amb l'argument `length`, bé copiant la longitud d'un altre vector, amb l'argument `along.with`

```
1 seq(from=10, length=30)
2 seq(from=10, length=30, by=2)
3 x <- 1:15
4 seq(from=10, along.with=x, by=2)
```

Les funcions, normalment, *retornen* un resultat, i per tant es pot assignar directament la crida a una funció a un objecte de **R**. Per exemple, la instrucció `m <- mean(x)` crida a la funció `mean()` amb l'argument `x`; aquesta retorna un número, i aquest número s'assigna a la variable `m`.

**28 §** Nosaltres podem definir funcions noves fàcilment: La sintaxi és

```
nom_funcio <- function( llista_parametres ) { cos_funcio }
```

on hem d'especificar el nom de la funció, la llista de paràmetres i el cos de la funció.

- El nom de la funció és arbitrari. Tan sols tingueu en compte que si hi ha algun altre objecte definit amb el mateix nom l'estarem canviant de valor.
- La *llista de paràmetres* són els noms que usarem dins de la funció per als arguments amb què cridem la funció. S'han de correspondre en el mateix ordre, excepte si usem noms d'argument en la crida.
- El *cos de la funció* són les instruccions que la funció executarà.

Comencem amb un exemple molt simple:

```
1 quadrat <- function(x){ x^2 }
```

<sup>7</sup>Tècnicament, són funcions diferents, encara que tinguin el mateix nom, però no cal entrar en aquest detall.

Aquesta funció eleva al quadrat allò que se li passi com a argument. Usem-la:

```
1 quadrat(3)
2 quadrat(1:5)
3 y <- 10
4 z <- quadrat(y)
5 print(y)
6 print(z)
```

Observeu que `quadrat(y)` no canvia el valor de `y`. Només l'eleva al quadrat i retorna el resultat, que nosaltres podem recollir en una altra variable. Proveu de fer `z <- quadrat(quadrat(y))`, i comproveu que enteneu el que fa.

La funció anterior té un sol argument. Fem-ne una amb dos arguments:

```
1 potencia <- function(x, p){ x^p }
2 potencia(2, 10)
3 potencia(1:5, 2)
4 y <- 1:10
5 potencia(y, 3)
```

Podem establir un *valor per omissió* d'alguns paràmetres. Per exemple, podem fer que la funció `potencia()` elevi al quadrat si no se li especifica el paràmetre `p`.

```
1 potencia <- function(x, p=2){ x^p }
2 potencia(4)
```

Si no donem valors per omissió, i els arguments no encaixen en quantitat i ordre amb els paràmetres, ens sortirà un error.

Com que els arguments tenen nom, al cridar la funció els podem especificar en qualsevol ordre, usant aquests noms.

```
1 potencia(p=2, x=10)
```

En els exemples anteriors, el cos de la funció només té una instrucció, que no és el més habitual. Si n'hi ha més d'una, les podem anar escrivint a la consola, però és molt més pràctic fer un script, que a més podrem guardar en un fitxer.

Feu aquest script, guardeu-lo en un fitxer i executeu-lo:

```
1 resum <- function(x){
2   x_bar <- mean(x)
3   sigma2 <- var(x)
4   paste("mitjana=", x_bar, ", variancia=", sigma2)
5 }
```

La funció `paste()`, com veieu, permet enganxar diferents objectes en una sola cadena i imprimir resultats de manera més endreçada.

**29 §** El valor que retorna una funció és, en principi, el resultat de la última instrucció de la funció. Però també es pot posar explícitament una instrucció `return()`, i de vegades és necessari. Ja veurem quan.

L'objecte retornat pot ser de qualsevol tipus (llista, gràfic, ...). Completem la funció anterior i fem que la sortida sigui una llista:

```

1 resum <- function(x){
2   x_bar <- mean(x)
3   sigma2 <- var(x)
4   m <- min(x)
5   M <- max(x)
6   list(mitjana=x_bar, variancia=sigma2, minim=m, maxim=M)
7 }

```

L'executem passant-li un vector com a paràmetre i assignem el resultat a un altre objecte:

```

1 resultats <- resum(5:15)
2 print(resultats)

```

**Exercici:** Gairebé sempre fem funcions per tenir-les guardades i reutilitzar-les quan calgui.

Guardeu el codi de la funció `resum()` en un script amb nom `MesuresBasiques.R` (per exemple). Tanqueu el fitxer i després tanqueu el **RStudio** (sense guardar el workspace!, com sempre).

Torneu a obrir el RStudio. A la consola, definiu un vector numèric qualsevol, que almenys tingui deu components, i calculeu el resum de mesures mitjançant la funció que hem creat abans. Tot això sense obrir cap fitxer.

**Exercici:** En un script hi pot haver definida més d'una funció. Un cop carregat amb `source()`, les tindrem totes disponibles.

Afegiu al fitxer `MesuresBasiques.R` el codi de la funció `potencia()`, i repetiu l'exercici anterior, calculant també la potència quarta del vostre vector.

## 9 Paquets

**30 §** Un *package* (paquet) de **R** és una col·lecció de funcions i/o dades. Quan el **R** s'engega, “carrega” en memòria uns quants paquets fonamentals, i per tant les seves funcions i dades les tenim disponibles de seguida. Totes les funcions que hem usat fins ara (excepte les que hem creat nosaltres mateixos) i les dades `airquality` que vam obrir al estudiar els data frames, estan en aquests paquets “precarregats”.

Per veure quins són aquests paquets, feu `search()`. Veureu una sortida com la següent:

```

1 [1] ".GlobalEnv" "package:stats" "package:graphics" "package:grDevices"
2 [5] "package:utils" "package:datasets" "package:methods" "Autoloads"
3 [9] "package:base"

```

Sense entrar en detalls tècnics, podeu pensar aquesta llista de la manera següent: La primera entrada, `.GlobalEnv`, és el “lloc on viuen” els objectes que *nosaltres* hem definit. És a dir, si feu ara `ls()` o `ls(.GlobalEnv)`, us ha de donar el mateix. En cas que el **R** no trobi un objecte que nosaltres referenciem, el buscarà en el paquet `stats`. Si feu `ls("package:stats")` veureu els objectes que hi ha definits en aquest paquet (per exemple, veureu `"var"`, que fa referència a la funció amb aquest nom que hem usat fa una estona).

**Opcional:** Podem “veure” el objecte `var` escrivint simplement el seu nom (sense els parèntesis). El que veurem és la definició de la funció `var()`. No obstant, ens



quedarem a mitges, perquè, després d'unes quantes instruccions de **R**, la funció acaba amb una instrucció `.Call`, que executa una funció escrita en llenguatge **C** i ja *compilada* (o sigui, traduïda a llenguatge màquina) i no es veu el seu codi. Moltes funcions bàsiques del **R** estan escrites en **C** i compilades. La seva execució és molt més ràpida perquè no s'han de traduir a llenguatge màquina mentre s'executen.

En cas que l'objecte que referenciem no estigui tampoc en `stats`, el buscarà en el següent paquet de la llista, i així successivament. L'últim que consulta és el paquet `base`. Aquí veureu noms que us seran familiars: `source`, `setwd`, `mode`, `mean`, `c`, ... En cas que no trobi aquí el nom, sortirà l'error `object not found`.

**Exercici:** Què passa doncs si nosaltres redefinim un objecte que està en un d'aquests altres paquets? Per exemple, feu `var <- function(x){print("jo no sóc la variància")}` i després feu `var(1:10)`.

Com podem recuperar la variància?!

**31 §** Podem obtenir una llista de tots els paquets que tenim instal·lats, demanar una petita descripció del paquet, o fer la llista de tots els objectes que conté, amb la seva descripció. Proveu:

```
1 library()
2 help(stats)
3 library(help=stats)
```

És important distingir entre tenir un paquet *instal·lat* i tenir-lo *carregat*. Per exemple, proveu `library(help=MASS)`. El paquet `MASS` va ser creat per acompanyar un llibre<sup>8</sup> (ja una mica antic), i conté funcions i dades d'exemple.

En el llistat que obteniu hi ha una secció `Functions` i una secció `Datasets`. Intenteu escriure a la consola el nom d'un dataset qualsevol, tal com vam fer amb `airquality`. No es troba. El mateix passarà si intenteu executar una de les funcions. El paquet està instal·lat perquè ve amb la instal·lació inicial del **R**, però ara mateix no el tenim carregat. De seguida veurem com carregar paquets instal·lats i com instal·lar nous paquets.

**32 §** Amb els paquets precarregats quan engeguem el **R**, n'hi ha un que s'anomena `datasets`. És l'únic d'aquests paquets que conté dades. I només conté dades; no funcions. Són dades "clàssiques" que han aparegut en articles i estudis científics; només tenen interès actual com a exemples per jugar i aprendre.

Com hem vist abans, un dels objectes que conté és `airquality`, que és un data frame. Un altre exemple és `AirPassengers`, amb dades del nombre de viatgers en vols internacionals, mensualment, entre 1949 i 1960. Mireu-lo escrivint el seu nom. Aquest no és un data frame, sinó una altra classe d'objecte que s'anomena *sèrie temporal* (*time series*), que no estudiarem en aquest curs de **R**. Podeu veure la classe d'un objecte amb la funció `class()`.

```
1 class(airquality)
2 class(AirPassengers)
```

<sup>8</sup>W. N. Venables, B. D. Ripley: *Modern Applied Statistics with S*, Springer 2002. Per què "S"? Resulta que **S** és el nom original del llenguatge, i **R** és una "implementació" del llenguatge **S**. Però no és moment d'entretenir-se en aquest detall.

Què passa si hi ha un objecte d'un paquet que té un nom igual a un objecte que nosaltres hem definit? Això és el que heu experimentat a l'exercici del paràgraf 30 §.

Com veiem amb la funció `search()`, el `.GlobalEnv`, que és on “viuen” els nostres objectes, va per davant de tots els altres paquets. En particular, per davant del paquet `datasets`. Si nosaltres fem, per exemple, `airquality <- 3` (feu-ho!), a partir d'aquest moment ja no podem veure l'objecte `airquality` del `datasets` (comproveu-ho!). El tornarem a veure si esborrem el nostre objecte fent `rm(airquality)`.

Anem a carregar el paquet `MASS`, que com hem vist conté dades i funcions. Feu

```
1 library(MASS)
2 data()
3 search()
```

La instrucció `data()` ens ensenya totes les dades que hi ha en els paquets que estan carregats. Se'ns obre una pestanya en el panel de l'editor on apareixen, separades per paquets (que ara són `datasets` i `MASS`), totes les dades disponibles.

La instrucció `search()` ens mostra que `.GlobalEnv` segueix estant per davant del nou paquet carregat. Per tant, igual que abans, si ara definim un objecte amb nom coincident amb un dels dels paquets carregats, el nostre té precedència. Si l'objecte el teníem definit abans que es carregui el paquet, la situació és la mateixa, i en aquest cas a més el **R** ens avisa. Anem a comprovar-ho. La primera instrucció serveix per revertir el `library(MASS)` (“des-carregar” el paquet). Després definim un objecte `cats` que existeix en el paquet `MASS`. Després tornem a carregar el `MASS`.

```
1 detach(package:MASS, unload=TRUE)
2 cats <- 5
3 library(MASS)
```

Veurem l'avís: `The following object is masked by_ '.GlobalEnv': cats`

El nostre objecte segueix tenint precedència i no podem usar el `cats` del `MASS`. Després de fer `rm(cats)` sí que hi podrem accedir. Proveu-ho.

Si és imprescindible accedir als dos objectes a la vegada, aleshores podem accedir al del paquet fent-hi referència com `MASS::cats`. Comproveu-ho.

Finalment, si fem `data(cats)`, el que estem fent és posar l'objecte `cats` dins el nostre lloc `.GlobalEnv`, i per tant, canviant el valor que poguéss tenir un objecte anteriorment definit amb el mateix nom.

**33 §** Per tenir una informació preliminar d'unes dades tenim la funció `help()`; per veure els “noms” que conté (per exemple, les columnes d'un data frame), podem fer `names()`; per visualitzar les  $n$  primeres o últimes files podem usar `head()` i `tail()`; la dimensió ens sortirà amb `dim()`.

```
1 help(cats)
2 dim(cats)
3 head(cats)
4 tail(cats)
5 names(cats)
```

Podem també accedir i carregar al `.GlobalEnv` dades d'un paquet sense carregar-lo tot:

```
1 detach(package:MASS, unload=TRUE)
2 help(eagles, package="MASS")
3 data(eagles, package="MASS")
4 print(eagles)
```

Ara tenim el data frame anomenat `eagles`, però cap altre objecte del paquet `MASS`.

**34 §** En alguna ocasió voldrem esborrar tots els objectes definits en una sessió per començar alguna cosa completament diferent sense interferències. La funció `rm()` admet un argument opcional anomenat `list`, una llista d'objectes; per altra banda, la funció `ls()` retorna una llista de tots els objectes en el `.GlobalEnv`. Per tant, una manera d'esborrar tots els nostres objectes és `rm(list=ls())`.

**35 §** Podem instal·lar nous paquets de dues maneres: La instrucció

```
1 install.packages("combinat")
```

instal·la el paquet anomenat `combinat`, que conté algunes fórmules de combinatòria. Com ja sabeu, amb `library(help=combinat)` veureu una descripció de les funcions que conté. És molt possible que el primer cop que instal·leu un paquet us preguntin si voleu crear una biblioteca personal. Podeu dir-li que no; no és important ara.

L'altra manera és usant el menú **Tools > Install Packages...**, i escrivint el nom del paquet en el diàleg que apareix.

Per carregar el paquet i tenir disponibles les seves funcions, podem fer `library(combinat)`, com ja sabem, i també usar la pestanya **Packages**, on apareixen tots els nostres paquets instal·lats, i marcar la casella del costat.

D'on estem baixant els paquets que instal·lem? Els paquets estan en *repositoris*, el més important dels quals és el CRAN (*Comprehensive R Archive Network*), que es troba a <http://cran.r-project.org>. És el que s'utilitza per omissió.

Podeu experimentar tot el que vulgueu instal·lant i carregant paquets. No passarà res irreversible.

## 10 Llegir i escriure fitxers amb dades

**36 §** Baixeu del Moodle el fitxer `Biblioteques.csv` i obriu-lo amb un editor de text qualsevol; per exemple, amb el **RStudio** mateix.

L'extensió `.csv` vol dir *comma-separated values*, i són fitxers de text pur, on típicament cada línia té una sèrie de camps, separats per uns caràcters que poden ser comes, punts i coma, blancs, tabuladors, etc. El fitxer que esteu veient, concretament, té els valors separats per punts i coma, i són dades sobre biblioteques de Catalunya, publicades al IDESCAT.

Per nosaltres és un format pesat de llegir, però al ser un fitxer de text pur, és perfecte per intercanviar informació i fer-la perdurable, perquè sempre hi haurà programes que el puguin llegir.

El que ens interessa es transformar aquestes dades a un objecte de la classe “data frame”, per poder treballar-lo dins el **R**. Això ho fem amb la funció `read.table()`. Abans d'aplicar-la ens hem de fixar en unes quantes coses del fitxer:

1. Hi ha una capçalera amb el nom dels camps? En aquest fitxer veiem que sí: hi ha una primera línia que és clarament diferent de la resta i que sembla que descriu què és cada camp. (Si l'editor amb què esteu veient el fitxer té activada la opció d'ajust de text (*text wrapping*), pot ser que veieu més d'una línia de capçalera, però en realitat és una sola; el caràcter (invisible) de final de línia només està darrera la paraula “social”.)
2. Quin és el símbol separador de camps? Típicament, trobem: comes, punts i coma, dos punts, barres verticals, espais en blanc, tabuladors (caràcter invisible, però diferent dels espais en blanc). En aquest cas, tenim punts i coma.
3. Quin símbol s'utilitza per a separar la part entera de la part decimal? Típicament, és un punt. Però també s'usa la coma. L'ús de l'apòstrof o “coma alta”, comú a Espanya fins els anys 1980's en l'escriptura manual, està en desús. Naturalment, si un fitxer `.csv` usa la coma en aquest sentit, aleshores no la pot usar com a separador. En el nostre fitxer, tot són nombres enters.
4. Hi ha valors perduts? Si hi camps buits, s'incorporaran al data frame com a **NA**. Però també poden haver-hi altres símbols que denotin que no es coneix un valor, com ara un guió (-), un interrogant (?), etc. En el nostre fitxer, a la última línia hi falta el primer valor.
5. Quina és la codificació del fitxer? Si el fitxer conté caràcters accentuat, saber la codificació del fitxer és important. Hi ha essencialment dues codificacions en ús actualment en la nostra zona geogràfica: “ISO-8859-1” i “UTF-8”. Si un caràcter accentuat ha quedat substituït per un símbol estrany, és que l'hem obert amb la codificació equivocada. En el **RStudio**, podem usar el menú **File > Reopen with encoding...** per obrir-lo suposant una altra codificació. En el nostre cas, el fitxer és veu bé en UTF-8.

Per carregar aquest fitxer al **R** com un data frame i visualitzar-lo, usarem les instruccions

```
1 bib <- read.table("Biblioteques.csv", header=TRUE,
2                 sep=";", encoding="UTF-8")
3 print(bib) # o bé View(bib)
```

L'argument `header=TRUE` indica que hi ha una capçalera (per omisió, el valor és `FALSE`); `sep` indica el caràcter separador (per omisió, és `sep=" "`, que s'interpreta com “espais en blanc”); finalment, especifiquem la codificació correcta. Si no s'especifica, el **R** agafa la que tingui el sistema operatiu per omisió; en el cas de Windows, és el ISO-8859-1, que és coneix també com a latin1, i s'especifica així en aquest paràmetre: `encoding="latin1"`.

Si veiem que en el fitxer hi ha per exemple els símbols - i ? per especificar valors perduts, afegirem l'argument `na.string=c("-", "?")`. Si no especifiquem res, només els camps en blanc passen a ser **NA** en el data frame.

**Exercici:** Observeu què passa quan especifiquem cadascun dels paràmetres anteriors de manera equivocada. És important veure el resultat per ajudar-nos a detectar ràpidament que alguna cosa no ha funcionat bé en una altra ocasió.

**Exercici:** Baixeu del Moodle el fitxer `2018_taxa_natalitat.csv`. Examineu-lo, i després feu l'assignació `natalitat <- read.table()` amb els arguments correctes.

**37 §** Com haureu vist en els exemples, el **R** afegeix un número de línia al data frame resultant. Podem fer que una de les variables actui com a nom de les files. Poseu `row.names="Literal"`, per exemple, en la lectura del data frame `bib`, per tal que el nom de les comarques passi a ser el nom de cada fila. No ens funcionaria `row.names="Codi"` perquè hi ha un valor que falta a la columna `Codi`.

Alguns fitxers de dades tenen a la capçalera un camp menys que en les files de dades. El conveni és que aleshores la primera columna actua com a nom de les files, i així ho interpreta la funció `read.table()`.

En lloc del nom d'un fitxer local podem posar directament l'adreça d'internet, la *url*. Per exemple:

```
1 url <-  
2 "https://vincentarelbundock.github.io/Rdatasets/csv/AER/EuroEnergy.csv"  
3 read.table(url, header=TRUE, sep=",", encoding="UTF-8")
```

En general, el format de les dades pot ser variat i el que vulguem fer amb elles també. Quan hi ha complicacions, convé mirar l'ajuda de la funció `read.table()`, que té moltes més opcions. També podem fer el següent:

**Exercici:** Feu **File > Import Dataset > From Text (readr)**. Escriviu la URL `https://www.idescat.cat/indicadors/?id=anuals&n=10370&col=1&f=ssv`, que ens baixa del IDESCAT les dades de matriculació a les universitats catalanes separades per sexes, des del curs 2000/01 fins al 2021/22. D'entrada, no es veuen correctament. Es tracta de provar diferents paràmetres en les **Import Options**. Canvieu **Delimiter** convenientment i feu que salti (skip!) les 10 primeres línies del fitxer, i ja veureu les dades bé. En el requadre **Code Preview** teniu el codi que heu d'executar per tal de baixar les dades i assignar-les a un data frame. Poseu el codi en un script.

Veureu que pel mig hi ha unes quantes línies més amb comentaris, i la resta de línies són les mateixes dades referides a Espanya. Afegiu una instrucció per quedar-nos amb les primeres 22 primeres files (les referides a Catalunya), assignant el resultat a un altre data frame.

Feu una gràfica de la columna `Total`.

Calculeu la suma de la columna `Total`. Si trobeu alguna dificultat, mireu l'ajuda de la funció `as.numeric()`.

**38 §** També podem fer el camí invers: guardar un data frame en un fitxer `.csv`. La funció que ho fa és `write.table()`. Proveu de guardar el data frame `natalitat` que heu creat a l'exercici del 36 § a un nou fitxer `naixements.csv`, però separant els camps per tabuladors en lloc de comes.

```
1 write.table(natalitat, file="naixements.csv",  
2             sep="\t", fileEncoding="UTF-8")
```

El fitxer `naixements.csv` estarà en el vostre directori de treball. Obriu-lo amb el **RSudio** o altre editor de text i observeu que el separador és ara el caràcter tabulador (invisible, però que en l'editor ocupa més d'un espai, típicament 2 o 4).

Paràmetres addicionals de `write.table()`:

- `quote`: Posant `FALSE`, no es guarden les cometes al voltant de les cadenes de text que en tinguin.

- `na`: Especifica la cadena de caràcters per representar les dades que falten (per omisió, és "NA")
- `dec`: Especifica el caràcter a usar com a punt decimal.
- `row.names`: Posant `FALSE`, no es guardaran els noms de les files.
- `col.names`: Posant `FALSE`, no es guardaran els noms de les columnes.

**Exercici:** Sabem que un data frame es pot indexar com una matriu. Useu-ho per crear un nou data frame on no hi siguin les columnes corresponents a `Any` i `Taxa_mil_hab` (que són inútils perquè totes les files tenen el mateix valor). Després guardeu-lo a un fitxer, usant la barra vertical `|` amb un espai a cada banda com a separador, traient-li les cometes als camps de text, i usant la coma en lloc del punt decimal. Comproveu el resultat en l'editor de text.

**39 §** El format de text pur `.csv` és el més segur per guardar i llegir dades. Tant el Excel com el LibreOffice permeten guardar en aquest format els seus fitxers nadius `.xls` i `.xlsx`, especificant codificació i separador.

Però també és possible, en principi, llegir/escriure data frames directament amb fitxers de Excel. Uns paquets moderns que fan aquesta feina (n'hi ha d'altres) són `readxl` i `writexl`, que possiblement haureu d'instal·lar.

Un cop instal·lats i carregats (vegeu el paràgraf 31 § i següents), podeu fer

```
1 as.data.frame(read_excel("lungcancer.xls"))
```

un cop baixat el fitxer `lungcancer.xls` que trobareu al Moodle. El `as.data.frame()` és necessari perquè l'objecte que retorna el `read_excel()` no és exactament un data frame, i s'ha de convertir.

## 11 Estructures algorísmiques

**40 §** El **R** és un llenguatge de programació complet, en el sentit que amb ell es pot codificar qualsevol *algorisme*, és a dir, qualsevol procediment de càlcul per produir un resultat a partir d'unes dades.

Hem vist que es poden executar diverses instruccions de **R** d'una sola tirada, fent un script, que es pot després cridar amb `source()`, des de la consola o des d'un altre script. I si el tenim obert en el panel d'edició del **RStudio**, prement el botó **Source** del mateix panel.

Però a més de la pura execució seqüencial, una rere l'altre, de les instruccions, disposem d'unes *estructures algorísmiques*, o *estructures de control de flux*. Anem a veure com són.

**41 §** Alternativa `if`:

```
1 if (condició) {  
2   # Instruccions si la condició és certa  
3 }
```

La `condició` és una expressió que dona un valor lògic, o sigui, `TRUE` o `FALSE`. Si és `TRUE`, s'executaran les instruccions dins de les claus. Si és `FALSE`, no s'executaran, i es continuarà executant el programa en la línia següent a la clau que tanca.

Obriu un script nou i aneu escrivint les instruccions dels exemples que anem veient en tot aquest apartat. Executeu-les seleccionant les línies i prement el botó **Run**. Per anar ràpid podeu fer **CTRL-RETURN** en el teclat.<sup>9</sup>

Exemple:

```
1 x <- 10  
2 if (x < 5) {  
3   print(x)  
4 }  
5 if (x > 5) {  
6   print(x + 1)  
7 }
```

La condició del primer `if` és `FALSE`, i no s'imprimeix res. La del segon és `TRUE`, i ens imprimeix el resultat de `x+1`.

Si només hi ha una instrucció a executar dins de les claus, podem no posar les claus. La posició de les claus també és arbitrària (podeu provar altres disposicions), però s'ha de procurar que el codi sigui llegible per nosaltres. La posició que veieu a l'exemple és força estàndard entre els programadors de **R**.

Així doncs, aquí podíem haver escrit també

```
1 x <- 10  
2 if (x < 5) print(x)  
3 if (x > 5) print(x + 1)
```

<sup>9</sup>En MacOS, **CMD-RETURN**.

## 42 § Alternativa `if ... else`:

```
1 if (condició){
2   # Instruccions si la condició és certa
3 } else {
4   # Instruccions si la condició és falsa
5 }
```

Si la condició és `TRUE`, s'executen només les instruccions que hi ha dins de les primeres claus. Si és `FALSE` s'executen només les instruccions de les segones claus. Després es continua executant el programa en la línia següent de la última clau. Exemple:

```
1 x <- 10
2 if (x %% 2 == 1) {
3   print(paste(x, " és imparell"))
4 } else {
5   print(paste(x, " és parell"))
6 }
```

L'operador `%%` calcula el residu de la divisió entera. Si el residu de dividir per 2 és 1, vol dir que el nombre és imparell, i ho escrivim a la pantalla. Si el residu no és 1 (i per tant és zero), el nombre és parell.

**Exercici:** És temptador estalviar-se les claus en aquest últim exemple, ja que cada acció té una sola instrucció. Proveu-ho i intenteu entendre per què no funciona.

**43 §** De vegades voldrem que la condició del `if(...)` sigui un vector amb més d'un element, i s'apliqui l'alternativa a cadascun dels elements. Això no funciona. Però podem utilitzar la funció `ifelse()` del paquet `base`.

Aquesta funció té tres arguments: La condició, l'acció a fer en cas que la condició sigui `TRUE`, i l'acció a fer en cas que sigui `FALSE`.

Per exemple, suposem que tenim un vector de números i volem fer una operació diferent amb ells segons siguin parells o imparells. El resultat serà un nou vector, amb tants components com tingui la condició, i contenint els resultats de les accions fetes. Proveu:

```
1 a <- 1:10
2 b <- ifelse(a %% 2 == 1, 3 * a + 1, a / 2)
3 print(b)
```

El segon i tercer arguments de `ifelse()` poden ser blocs d'instruccions entre claus, o també la crida a una funció. Un bloc entre claus es comporta com una funció, en el sentit que retorna el que retorna la seva última instrucció. Ho podeu comprovar amb aquest codi, que dona el mateix resultat que l'anterior:



```
1 a <- 1:10
2 b <- ifelse(a %% 2 == 1,
3   {
4     c <- 3 * a
5     c + 1
6   },
7   {
8     c <- a
9     c / 2
10  }
11 )
12 print(b)
```

#### 44 § Repetició `for`:

```
1 for(i in x){
2   # Instruccions a repetir
3 }
```

essent `x` un vector. La repetició s'utilitza per executar un mateix bloc d'instruccions per a cadascun dels elements del vector. El vector pot ser de qualsevol tipus. Un cop acabats els elements, l'execució del programa continua en la instrucció següent de la clau que tanca. Exemple: Calculem el quadrat de tots els números del 1 al 15 i imprimim els residus de dividir el quadrat per 5.

```
1 x <- 1:15
2 for (i in x) {
3   j <- i^2
4   print(j %% 5)
5 }
```

Si es pot aplicar directament al vector l'operació que volem, és millor que usar un `for`, perquè és molt més ràpid. En aquest cas, `(1:15)^2 %% 5` dona el mateix resultat.

Un altre exemple amb vectors de mode caràcter:

```
1 x <- c("pomes", "taronges", "maduixes", "llimones")
2 for (i in x) {
3   magraden <- paste("M'agraden les", i)
4   print(magraden)
5 }
```

#### 45 § Iteració `while`:

```
1 while (condicio) {
2   # Instruccions a iterar
3 }
```

Les instruccions dins les claus es van repetint, mentre la condició sigui `TRUE`. Quan sigui `FALSE`, el programa continua en la instrucció següent després de la clau que tanca. És necessari assegurar-se que la condició sigui falsa en algun moment, perquè si no el

nostre programa no acabaria mai. Afortunadament, des del **RStudio** podem aturar un programa en cas de necessitat, prement la tecla **ESCAPE**.

Exemple simple:

```
1 i <- 1
2 while (i < 10) {
3   print(i)
4   i <- i + 1
5 }
```

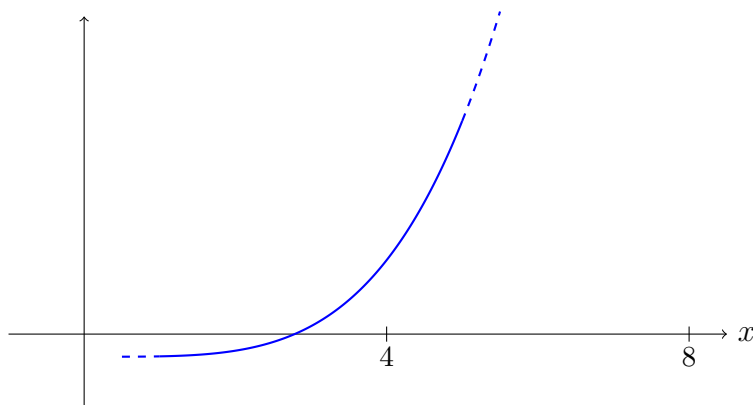
A la primera iteració, la variable `i` val 1, i per tant compleix la condició. S'imprimeix el seu valor, i després la incrementem en una unitat. Tornem a avaluar la condició. Ara `i` val 2, i la condició segueix essent **TRUE**. S'imprimeix el 2, i després `i` passa a valer 3. I així successivament, fins la iteració en què s'imprimeix un 9, i la variable `i` passa a valer 10. Ara la condició dóna **FALSE**, i ja no s'executen més iteracions.

Aquest no és un bon exemple d'aplicació de l'estructura `while`, perquè sabem d'entrada quantes iteracions farem. En tal cas, és més lògic i elegant utilitzar la repetició `for`:

```
1 for (i in 1:9) print(i)
```

L'estructura `while` és més propi utilitzar-la quan no sabem quants cops es repetiran les instruccions. Farem un exemple una mica més elaborat per il·lustrar-ho:

Suposem que volem la solució de l'equació  $x^4 - 60 = 0$ . Veiem substituint que la funció  $x^4 - 60$  dóna negatiu per  $x = 0$ , i dóna positiu per  $x = 8$ . A més, és una funció creixent per  $x > 0$ . Per tant, hi ha un únic punt  $x$  entre 0 i 8 on la funció creua l'eix horitzontal, i aquest punt és la solució de l'equació.



Podem trobar aquest punt amb el procediment següent: Avaluem la funció en el punt mig de l'interval  $[0, 8]$ . Si dóna negatiu, sabem, pel mateix argument, que la solució estarà en l'interval  $[4, 8]$ ; si dóna positiu, estarà en  $[0, 4]$ . I tornem a iterar. L'interval on sabem que estarà la solució serà la meitat de gran a cada iteració. Ens aturarem quan la diferència entre l'extrem dret i l'extrem esquerre de l'interval sigui més petit que una tolerància que nosaltres fixarem. Al final agafarem el punt mig de l'últim interval i podrem afirmar que el resultat és correcte amb un error més petit que la tolerància.

```
1 esquerra <- 0
2 dreita <- 8
3 tolerancia <- 1e-5
4 while (dreita - esquerra > tolerancia) {
5   x <- (esquerra + dreita) / 2
6   if (x^4 - 60 < 0) {
7     esquerra <- x
8   } else {
9     dreita <- x
10  }
11 }
12 x <- (esquerra + dreita) / 2
13 print(x)
```

Observeu que:

- Dins del `while` tenim un `if...else`. Podem tenir diferents estructures unes dins de les altres, fins al nivell de profunditat que calgui.
- No sabem d'entrada quantes iteracions farem. Cal anar avaluant la condició cada cop per saber si hem d'entrar dins el `while` o no.
- `1e-5` és una manera equivalent, habitual en informàtica, d'escriure `10(-5)`, o sigui `0.00001`.
- Si voleu veure com va evolucionant el valor de l'aproximació `x` a cada iteració, poseu un `print(x)` com a última instrucció dins el `while`.

**46 §** Hi ha dues instruccions especials que es poden posar dins de l'estructura `while` i que modifiquen el flux de l'execució. La primera és `break`, que atura les iteracions immediatament, sense acabar la que s'estigui fent. La segona és `next`, que atura la iteració actual i passa a la següent.

Es poden usar dins de `for`, `while` i l'estructura `repeat`, que veurem després.

Per exemple, suposem que en el càlcul de la solució de  $x^4 - 60 = 0$  que hem fet, volem introduir una segona tolerància, de manera que si la diferència entre  $x^4 - 60$  i zero és més petita que  $10^{-2}$ , l'aproximació ja ens val i aturem el càlcul.

```
1 esquerra <- 0
2 dreita <- 8
3 tolerancia <- 1e-5
4 tolerancia2 <- 1e-2
5 while (dreita - esquerra > tolerancia) {
6   x <- (esquerra + dreita) / 2
7   if (x^4 - 60 < 0) {
8     esquerra <- x
9   } else {
10    dreita <- x
11  }
12  if (abs(x^4 - 60) < tolerancia2) break
13 }
14 print(x)
```

**Exercici:** En aquest cas, l'ús del `break` era fàcilment evitable. Podem posar les dues condicions d'aturada juntes a la condició del `while`, amb l'operador de conjunció `&`. Feu-ho.

47 § Iteració `repeat`:

```

1 repeat{
2   # Instruccions a iterar fins que arribi un break
3 }

```

Les instruccions dins les claus es repeteixen infinitament en bucle. Per tant és imprescindible que hi hagi en algun lloc una instrucció `break` que ens faci sortir. Exemple senzill: volem trobar el nombre enter  $k$  més petit tal que  $k^3 > 50\,000$ .

```

1 k <- 1
2 repeat{
3   cub <- k^3
4   if (cub <= 5 * 10^4) {
5     print(paste(cub, "és més petit que 50 000. Seguirem iterant."))
6     k <- k + 1
7   } else {
8     print(paste(cub, "és més gran que 50 000. Sortim del bucle."))
9     break
10  }
11 }
12 print(paste("El número és:", k))

```

Les estructures iteratives `while` i `repeat` s'assemblen, i el que es pot fer amb una es pot fer amb l'altre, canviant el que calgui canviar. Una diferència fonamental és que les instruccions del `while` pot ser que no s'executin mai, si la condició no es compleix d'entrada, mentre que les instruccions del `repeat` s'executen segur una vegada, almenys fins el `break`.

Repetim l'exemple de trobar la solució de  $x^4 - 60 = 0$  (la última versió) usant `repeat` en comptes de `while`. Observeu que és natural posar la condició a l'inrevés, i al final del bloc.

```

1 esquerra <- 0
2 dreta <- 8
3 tolerancia <- 1e-5
4 tolerancia2 <- 1e-2
5 repeat{
6   x <- (esquerra + dreta) / 2
7   if (x^4 - 60 < 0) {
8     esquerra <- x
9   } else {
10    dreta <- x
11  }
12  if (dreta - esquerra <= tolerancia |
13     abs(x^4 - 60) <= tolerancia2) {
14    break
15  }
16 }
17 print(x)

```

**48 § Exercici:** Començant en un nombre enter positiu qualsevol  $x$ , volem fer:  $3x + 1$  si  $x$  és imparell, i  $x/2$  si  $x$  és parell. Al resultat, li aplicarem el mateix, i així iterativament fins que surti un 1. Per exemple, si comencem amb  $x = 17$ , obtenim la successió

17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1

- Useu una estructura iterativa `while` o `repeat` per fer aquesta iteració fins arribar a  $x = 1$ , amb l'alternativa `if` dins del bucle per distingir cada cop si el nombre és parell o imparell. Fixeu de moment  $x = 17$  com a valor inicial per comprovar que us surt el resultat esperat, imprimint tots els valors pels quals passeu. Guardeu el codi en un script i executeu-lo amb [Source](#).
- Afegiu un comptador que ens digui al final quants elements té la successió, imprimint el missatge `Començant amb x=17, hi ha ... elements`. Comproveu que surten 13 elements en aquest cas.
- Volem provar com a valors inicials tots els enters entre 2 i 100. Afegiu una estructura `for` per anar variant el valor inicial. Podeu esborrar o deixar comentat el `print()` de tota la successió perquè no sigui tan llarga la sortida. Heu de fer que el missatge de la longitud de la successió sigui `Començant amb x=..., hi ha ... elements`
- Volem saber, de totes aquestes successions, quina és la més llarga. Una manera de fer-ho és, cada vegada que calculem la longitud d'una successió, comparar-la amb la longitud més gran obtinguda fins el moment; i si és superior, guardar el valor inicial d'aquella iteració i actualitzar la longitud més llarga trobada. Feu que imprimeixi al final `Entre 2 i 100, la successió més llarga és la que comença per x=... i té ... elements`. Quan veieu que funciona bé, podeu esborrar o comentar els `print()` de la longitud de cada successió.
- Finalment, construïm una funció que faci aquests càlculs, i que tingui dos arguments: de quin fins a quin número volem agafar com a valors inicials de les successions (el paper que fan el 2 i el 100 fins ara). Anomenant `f` (o com vulgueu) a la funció, només cal ficar tot el codi que tenim dins de

```
1 f <- function(primer, ultim) {  
2   ...  
3 }
```

i substituir les aparicions de 2 i 100 per les variables `primer` i `ultim`.

Un cop heu fet [Source](#), comproveu des de la consola que `f(2, 100)` dóna el que ja sabem, i proveu altres valors, com ara `f(100, 1000)`.

- (Opcional) Per tal que la funció anterior sigui perfecta, hauria de controlar que l'usuari no entri valors invàlids. Per tant, hauria de començar amb la comprovació que `primer` i `ultim` són números enters, que `primer > 1` i que `primer < ultim`. Comprovar que s'ha introduït un nombre enter no és tan fàcil; una altra opció és assegurar-se'n fent `x <- as.integer(x)`, que descarta la part decimal, si n'hi ha. Però, abans que res, caldria comprovar que realment s'han introduït números, i no lletres o altres coses. Com veieu, és complicat filtrar tot el que pot anar malament.

## 12 Les funcions `apply()` i família

**49 §** La família “`apply`” és una col·lecció de funcions que faciliten certes manipulacions de arrays, llistes i data frames, tot evitant haver d'usar estructures repetitives tipus `for`.

La funció bàsica és `apply()` que actua sobre les files o columnes d'una matriu, o més en general, sobre qualsevol “llesca” d'un array de més dimensions.

Obriu un fitxer nou amb el **RStudio**, i feu:

```
1 x <- array(1:12, dim=c(3,4))
2 print(x)
3 apply(x, 1, sum)
4 apply(x, 2, sum)
```

Hem “aplicat” la funció `sum()` (suma dels elements d'un vector) a cadascuna de les files de la matriu (quan el segon argument és 1), i a cadascuna de les columnes (quan el segon argument és 2). Podem utilitzar altres funcions, com ara `mean()` o `var()`, que també retornen un número per cada fila o columna de la matriu.

Apliquem-ho a un array de més dimensions:

```
1 x <- array( 1:24, dim=c(3,4,2))
2 print(x)
3 apply(x, 1, sum)
4 apply(x, 2, sum)
5 apply(x, 3, sum)
6 apply(x, c(1,2), sum)
```

Quan posem `1`, estem sumant per “files” (la primera dimensió). Observeu que la primera fila està formada per `1, 4, 7, 10, 13, 16, 19, 22`, i la suma és 92.

Quan posem un `2`, estem sumant per “columnes” (la segona dimensió). La primera columna, per exemple, té els sis elements `1, 2, 3, 13, 14, 15`.

Quan posem un `3`, estem sumant per la tercera dimensió. La primera “llesca” del array en aquesta dimensió té els números del 1 al 12 i suma 78. La segona llesca té els números del 13 al 24, i suma 222.

També podem fer `apply(x, c(1,2), sum)`, per exemple. El resultat serà la suma per a cada combinació de (fila, columna); per tant una matriu. Cada element d'aquesta matriu és la suma de dos números. Per exemple, el primer element és la suma de 1 i 13. L'últim element és la suma de 12 i 24.

**50 §** També podem aplicar `apply()` a un data frame. Per exemple, usant les dades `mtcars` del paquet `datasets`, amb

```
1 mtcars
2 apply(mtcars, 2, mean)
```

calculem la mitjana de totes les variables d'aquest data frame. El resultat és un vector “amb noms”, els mateixos que obteniu si feu `names(mtcars)`.

La funció que apliquem també pot retornar més d'un número. Per exemple, apliqueu la funció `summary()`. El resultat ara, en lloc d'un vector, és una matriu “amb noms”.

**51 §** La funció `lapply()` s'aplica a llistes o vectors i retorna una llista. No cal indicar-li cap “dimensió”:

```
1 lapply(mtcars, mean)
```

Aquí l'hem aplicada a un data frame, però recordeu que el **R** veu els data frame també com a llistes.

Apliqueu també la funció `summary()` amb `lapply()`. Us ha de sortir una llista, els elements de la qual són vectors amb noms.

La funció `sapply()` fa el mateix que `lapply()`, però el resultat el dona en forma de vector o matriu en lloc de llista. Si ho apliquem a `mtcars`, veiem que el resultat és idèntic a `apply()`, però sense haver d'especificar la dimensió. Proveu-ho.

**52 §** La funció `tapply()` s'aplica sobre un vector, partint-lo en trossos segons una altra variable, per tal d'aplicar la funció del tercer argument per separat a cadascun dels trossos.

Seguim agafant com exemple les dades `mtcars`. La segona variable `cyl` és el nombre de cilindres del motor, que poden ser 4, 6 o 8. Ens agradaria calcular la mitjana del consum (primera variable, `mpg`), i el temps per fer el quart de milla (variable `qsec`), separant per nombre de cilindres. Això ho podem fer simplement amb

```
1 tapply(mtcars$mpg, mtcars$cyl, mean)
2 tapply(mtcars$qsec, mtcars$cyl, mean)
```

En lloc de la “dimensió” hi posem el vector que ens determina les diferents classes. Proveu també de canviar `mean()` per `summary()`.

**53 §** La funció que posem com a tercer argument pot ser una funció estàndard del **R** o una funció que nosaltres definim. Podem per exemple aprofitar això per convertir la variable `mpg` (mil·les per galó) a litres per cada 100 quilòmetres, que és una mesura més comuna a Europa.

```
1 mean_consum_EU <- function(x) {
2   y <- 282.4811/x
3   mean(y)
4 }
5 tapply(mtcars$mpg, mtcars$cyl, mean_consum_EU)
```

## 13 Entorns

**54 §** Recordeu que dèiem a l'Apartat 9, paràgraf 30 §, que la funció `search()` retorna la llista de paquets que tenim carregats, en un ordre que és en el que “busca” un objecte quan nosaltres hi fem referència. Primer el busca en el “lloc” on nosaltres definim objectes, que és el `.GlobalEnv`; si no el troba busca en el paquet `stats`; si no el troba el busca en el següent de la llista, etc.

Anem a ser ara una mica més precisos amb la terminologia. La llista que veieu al fer `search()` és una llista de *entorns* (*environments*), començant per l'entorn “global” (el `.GlobalEnv`).

Feu `rm(list=ls())` per esborrar tot el que tenim en el `.GlobalEnv`, i executeu les instruccions següents:

```
1 x <- 3
2 y <- 5
3 quadrat <- function(z) { z^2 }
4 ls()
5 environment()
```

Hem llistat els objectes de l'entorn global. Apareixen `x`, `y`, `quadrat`, que són els que hem definit. No apareix `z`, que és un objecte intern de la funció `quadrat()`; no està en el nostre entorn global. La instrucció `environment()` ens diu que estem en l'entorn global (`R.GlobalEnv` és el mateix que `.GlobalEnv`)

La funció `quadrat()` crea un nou entorn, que és on viu `z`. És un entorn *fill* de `.GlobalEnv`; `.GlobalEnv` és el seu entorn *pare*. Torneu a eliminar tots els objectes de l'entorn global amb `rm(list=ls())`, i feu un script amb aquest codi:

```
1 f <- function(x){
2   g <- function(y){
3     print("Dins de g:")
4     print(environment())
5     print(ls())
6   }
7   g(1)
8   print("Dins de f:")
9   print(environment())
10  print(ls())
11 }
12 f(0)
13 print(environment())
14 print(ls())
```

Analitzem atentament el que hem fet:

- Al nivell més alt (entorn global), hem definit una funció `f`, que després hem cridat amb `f(0)`.
- Dins la funció `f`, hem definit una funció `g`, que després hem cridat amb `g(1)`.
- Dins de la funció `g`, hem escrit el nom del seu entorn i els objectes que hi ha en aquest entorn. Això és el primer que surt imprès a la sortida. El nom de l'entorn s'assigna automàticament i no és important. Només interessa que *no és* l'entorn global. Conté un sol objecte, que és el paràmetre `y`, amb valor 1, com podeu comprovar si afegiu la instrucció `print(y)` allà dins.
- Un cop hem retornat de `g`, continuem amb `f`. Escrivim el nom del seu entorn i els objectes que conté. Veiem que és un entorn diferent del de `g` i diferent del global, i que conté el paràmetre `x`, que val 0, i la funció `g`.
- Finalment, les dues últimes instruccions del programa escriuen l'entorn, que ara sí és el global, i l'únic objecte que conté, que és la funció `f`.

Deduïm que des de l'entorn global no podem cridar a la funció `g` ni accedir als objectes `x`, `y`.



**55 §** Un cop més esborreu tots els objectes de l'entorn global: `rm(list=ls())`. Com que estem esborrant `f`, també estem eliminant el seu entorn i en conseqüència també la funció `g` i el seu entorn. Feu ara:

```
1 f <- function(x){
2   g <- function(y){
3     c <- 30
4     a+b+c
5     print("Dins de g:")
6     print(ls())
7   }
8   b <- 20
9   a+b
10  g(1)
11  print("Dins de f:")
12  print(ls())
13 }
14 a <- 10
15 f(0)
16 print(ls())
```

Observeu ara que, apart dels paràmetres `x` i `y` de les funcions, tenim unes variables `a`, `b`, `c` definides en diferents entorns.

La variable `a` està definida a l'entorn global, i és accessible des de dins de `f` i des de dins de `g`. Aquestes variables s'anomenen *globals* i estan accessibles a tot arreu en el programa des del moment en què estan definides.

Però cal tenir en compte que la globalitat d'una variable s'ha de pensar des del punt de vista de cada funció. Des del punt de vista de `g`, la variable `b` també és global, en el sentit que està definida en un entorn superior.

En canvi, des de l'entorn de la funció `f` no es pot accedir a la variable `c`, i des de l'entorn global no es pot accedir a les variables `a` i `b`. Podeu provar-ho i veureu com surt un error. Aquestes variables són *locals*. Concretament, la variable `c` és local de l'entorn de `g`; la variable `b` és local de l'entorn de `f`.

**56 §** Què passa si hi ha noms de variables de diferents entorns que coincideixen? Després d'esborrar-ho tot amb `rm(list=ls())`, fem:

```
1 f <- function(){
2   a <- 20
3   g <- function(){
4     a <- 30
5     print(a)
6   }
7   g()
8   print(a)
9 }
10 a <- 10
11 f()
12 print(a)
```

Després de l'assignació a nivell global `a <- 1`, entrem a la funció `f` i canviem el valor de `a`. Entrem a `g` i el tornem a canviar. No obstant, al tornar als entorns exteriors, la variable segueix tenint el valor que tenia abans. Concloem que:

- Una variable global és accessible, es pot usar, i es pot canviar, dins d'una funció.
- Al acabar la funció, la variable global no ha canviat.

La manera fàcil d'entendre-ho és pensar que l'entorn de la funció crea una còpia apart de la variable i treballa amb ella com a variable local. No toca la variable global.

**57 §** No és impossible canviar una variable global des de dins d'una funció. Cal usar l'operador de “superassignació” `<<-`. Esborrem tot altre cop i fem:

```
1 f <- function(){
2   a <<- 20
3   print(a)
4 }
5 a <- 10
6 f()
7 print(a)
```

Específicament, aquesta assignació amb fletxa de dues puntes el que fa és: Buscar la variable en el entorn pare; si no la troba, el busca en el següent nivell; i així successivament fins a l'entorn global. Si a l'entorn global la variable no existeix, la crea *en aquest entorn global*.

**Exercici:** Intenteu predir què passarà amb aquest codi:

```
1 rm(list=ls())
2 f <- function(){
3   g <- function(){
4     a <<- 30
5     print(a)
6   }
7   a <- 20
8   g()
9   print(a)
10 }
11 a <- 10
12 f()
13 print(a)
```

Els informàtics aconsellen sempre no usar variables globals dins de les funcions, en cap llenguatge de programació. Hi ha bones raons. Una funció ben feta i reutilitzable hauria d'interaccionar amb el seu entorn pare només a través dels arguments que se li passen al cridar-la, i del valor que retorna. Per tant, no hauria d'utilitzar variables globals i menys encara modificar-les. No obstant, des del punt de vista pràctic, aquesta possibilitat ens permet acabar una prova ràpida, que ja arreglarem després, si cal.

**58 §** Així com diem que l'entorn definit per una funció és fill de l'entorn global `.GlobalEnv`, podem dir que `.GlobalEnv` és fill de l'entorn `package:stats`, en el sentit que si no troba un objecte en `.GlobalEnv`, el busca en `package:stats`. La llista donada per `search()` ens proporciona la cadena d'entorns en l'ordre de fill a pare, fins arribar a `package:base`, que no té pare.

Quan carreguem un paquet, per exemple `library(MASS)`, aquest paquet queda inserit com a pare directe de `.GlobalEnv` i fill de `package:stats`. Comproveu-ho. Si carreguem un

altre paquet, quedarà entre `.GlobalEnv` i `package:MASS`. El paquet més recent agafa el paper de pare de `.GlobalEnv`. Comproveu-ho, carregant qualsevol altre paquet dels que teniu llistats a la pestanya **Packages**.

**Exercici:** La funció `var()` està definida en el paquet `stats` i calcula la variància dels valors d'un vector. La funció `sd()` (desviació tipus) està definida en el mateix paquet, essencialment com l'arrel quadrada de la variància: `sd(x) <- sqrt(var(x))`.

Definiu un vector `x` amb cinc números qualssevol i calculeu la seva variància i desviació tipus; després feu `var <- 4`. Què creieu que passarà ara al fer `sd(x)`? Comproveu-ho. (Solució al peu de pàgina.<sup>10</sup>)

**59 §** Dins d'un script es pot cridar la funció `source(...)`, que ja hem usat des de la consola. Quan l'execució del script arriba a aquesta funció, llegeix el fitxer que se li indica, l'executa i després continua en la línia següent del script. És gairebé com si copiéssim i enganxéssim el codi del fitxer en el nostre script.

Hi ha només un petit detall, però important, a tenir en compte: El fitxer que es llegeix s'executa en l'entorn global, i no en l'entorn en què l'estem llegint. Si volem que s'executi en l'entorn local, com si realment féssim un copiar/enganxar, aleshores cal fer `source(..., local=TRUE)`.

## 14 Precisió en els càlculs

**60 §** La funció `options()` permet veure i modificar algunes opcions molt bàsiques del funcionament del **R**. Per exemple, si no us agrada el símbol `>` que apareix a la consola quan espera una instrucció (el *prompt*), el podeu canviar per alguna altra cosa. Proveu per exemple `options(prompt="Què vols? ")`.

L'opció més interessant de conèixer és `digits`, que per omissió té el valor 7. Això vol dir que els resultats dels càlculs surten amb set xifres significatives. Proveu:

```
1 x <- 10.87667748768532147
2 print(x)
```

Només ens dona 7 xifres, arrodonint correctament. Però la resta no s'ha perdut:

```
1 options(digits=10)
2 print(x)
3 options(digits=15)
4 print(x)
```

Cada cop arrodoneix correctament a les xifres que se li donin (comptant part entera i part decimal). El valor màxim que podem posar és `digits=22`, però en aquest cas no ens donarà més de 17 xifres, i de fet la última pot estar mal arrodonida. Comproveu-ho.

El que succeeix és que el **R** emmagatzema els números en un espai limitat de memòria. Concretament, usa una *mantissa de 53 bits*<sup>11</sup>, que correspon aproximadament a 16 xifres bones en base 10.

<sup>10</sup>El que heu comprovat és que la funció `sd()` segueix funcionant bé, potser en contra de la nostra intuïció en aquest moment. Sense entrar en detalls tècnics, el que passa és que `sd()` està definida dins un "espai de noms" (*namespace*) que la protegeix de canvis en els objectes que la defineixen.

<sup>11</sup>també anomenada *mantissa de doble precisió*.

```
1 options(digits = 20)
2 100 / 998
```

El **R** ens dona el resultat amb 20 xifres, però les 3 últimes estan malament! Ho podeu comprovar amb el **SageMath**, que sí que fa el càlcul amb nombres enters amb total precisió, amb la instrucció `(100/998).n(digits=20)`.

Normalment, aquesta imprecisió del **R** és un problema molt menor. Un error a la dissetena xifra significativa vol dir un error relatiu de com a molt  $10^{-16}$ , que sol ser negligible.

**61 §** De tota manera, pot haver-hi situacions en què vulguem treballar amb nombres el més exactes possible. Per exemple, suposem que volem saber de quantes maneres es pot ordenar un conjunt de 30 elements. Ho podem calcular amb la funció `factorial()`:

```
1 factorial(30)
```

El resultat és un nombre enter de 33 xifres. Però encara que hàgim posat el màxim `options(digits=22)`, només obtenim les 17 xifres més significatives, i la última segurament no és bona. En efecte, amb **SageMath** obtindreu `26525285981219105863630848000000`. L'última xifra del resultat del **R** està equivocada, i si s'ha d'arrodonir a 16 xifres, la setzena hauria de ser un 1, no un zero, cosa que no podem deduir de la sortida.

Anem a veure un altre exemple. Farem una funció que trobi les solucions reals de l'equació de segon grau  $ax^2 + bx + c = 0$ .

```
1 eqGrau2 <- function(a, b, c) {
2   discriminant <- b^2 - 4 * a * c
3   if (discriminant >= 0) {
4     radical <- sqrt(discriminant)
5     print(c(-b - radical, -b + radical) / (2 * a))
6   } else {
7     print("no hi ha solucions reals")
8   }
9 }
10
11 a <- 1
12 b <- -5
13 c <- 6
14 eqGrau2(a, b, c)
```

El resultat és perfectament correcte. Les solucions exactes són  $x = 2$ ,  $x = 3$ , i ens dona el mateix tant si ho fem amb 7 dígitos com amb 17 dígitos.

Si dividim per 3 tots els coeficients de l'equació, el resultat hauria de ser el mateix. Però...

```
1 a <- 1/3; b <- -5/3; c <- 6/3
2 options(digits = 7)
3 eqGrau2(a, b, c)
4 options(digits = 17)
5 eqGrau2(a, b, c)
```

Ups! Ara el resultat és correcte quan demanem 7 dígitos, i no ho és quan demanem 17. La setzena xifra ja està malament. Si feu `1000*eqGrau2(a, b, c)` i `100000*eqGrau2(a, b, c)`, veureu més coses estranyes.

Aquests resultats són conseqüència del fet que el **R** no pot guardar exactament números com  $1/3$  o  $-5/3$ . Els guarda en un cert nombre de bits (zeros i uns, en sistema binari), arrodonint correctament. Igual que no és possible representar 'un terç' en una quantitat finita de decimals. Quan després fem operacions amb aquests números, els errors de representació interna es van propagant, i a la llarga es poden acumular i ser molt més grans.

Un últim exemple, més dramàtic: Useu la funció `eqGrau2()` per resoldre l'equació  $x^2 - 1000.001x + 1 = 0$ . Les solucions exactes són  $x = 0.001$  i  $x = 1000$ . Comproveu que surten moltes menys xifres bones. El quart decimal d'una de les solucions està completament malament.

L'explicació és que certs valors dels coeficients (aquells pels quals  $b^2$  és molt més gran que  $4ac$ ), provoquen la resta de dos números molt propers, i a conseqüència d'això es perden moltes xifres significatives. S'ha d'evitar restar valors semblants. El programa que hem fet, usant la fórmula clàssica, no és un bon programa per resoldre qualsevol equació de segon grau.

**62 §** Si necessitem fer càlculs amb més precisió, tenim un paquet de **R**, anomenat `Rmpfr`, que ens ho permet. Està basat en la biblioteca *Multiple Precision Floating-Point Reliable Library* del llenguatge **C**, que permet fer càlculs amb precisió arbitrària.

Instal·leu i carregueu aquest paquet. Feu aleshores:

```
1 factorial(mpfr(30, prec = 120))
```

el resultat ara és exacte. L'argument `prec` indica la precisió en bits amb què volem fer el càlcul. Recordeu que hem dit en el 60 § que en principi el **R** utilitza sempre 53 bits.

Com sabem quin és el valor de `prec` que hem de posar? Si `bits` és el que hem de posar, i `dígit`s són les xifres significatives desitjades, les dues quantitats estan relacionades per la fórmula

$$2^{\text{bits}} = 10^{\text{dígit}}$$

d'on es dedueix que

$$\text{bits} = \text{dígit} \times \log_2(10)$$

En aquest cas concret, veiem de la resposta aproximada que necessitarem 33 dígit. Calculem `33*log2(10)` i veiem que 120 bits és més que suficient.

**Exercici:** Amb una sola instrucció, feu que el **R** ens doni els factorials correctes i exactes dels 24 primers nombres naturals. Podeu usar `prec=120`, que ja hem vist que és suficient per aquestes quantitats.

**Exercici:** Calculeu de quantes maneres exactament poden quedar ordenades les 52 cartes d'una baralla, calculant primer un valor raonable de `prec`.



## 15 Rmarkdown

**63 §** Els resultats d'una anàlisi estadística no són només números i gràfics. El destinatari dels informes estadístics és freqüentment algú que ha de prendre decisions. Cal per tant *redactar*, desenvolupar la història que els resultats expliquen, de manera correcta i el més estètica possible.

Veurem en una altra part de l'assignatura que el  $\text{\LaTeX}$  ens ajuda a produir documents de gran qualitat en format PDF, on hi podem incloure fórmules, taules, gràfics, i text ordinari.

Però hi ha una manera més ràpida i fàcil de crear informes força presentables, sense sortir del **RStudio**, en format HTML, que permet visualitzar-los en un navegador d'internet com qualsevol pàgina web. Necessitem instal·lar i carregar el paquet de **R** anomenat `rmarkdown`. Feu-ho.

Un cop instal·lat aquest paquet, podrem crear, en el **RStudio** mateix, fitxers que barregen:

- el text que volem explicar,
- unes marques per assenyalar títols d'apartats, llistes, fórmules, ... ,
- el codi en **R** que volem executar.

**64 §** Veiem l'exemple-mostra que porta el mateix **RStudio**: En el menú, feu **File > New File > R Markdown...** . Veureu un diàleg amb opcions. Podeu deixar-lo tal com està i acceptar. Us crearà un nou fitxer amb una plantilla de document. Feu **File > Save As...** i guardeu-lo amb un nom acabat en `.Rmd`, com ara `test.Rmd`.

Ara premeu el botó **Knit** en el panel de l'editor. Se us obrirà una finestra apart, amb el document processat. Aquesta finestra ens està mostrant un fitxer escrit en HTML, que ara tenim guardat com `test.html` en el mateix directori. Ens el mostra com ho faria Chrome, Firefox o qualsevol altre navegador d'internet. De fet, el podem obrir amb el nostre navegador tocant el botó **Open in Browser** d'aquesta finestra. Heu produït una pàgina web senzilla, però prou agradable de llegir, sense necessitat de saber HTML.

Tornant al **RStudio**, anem a examinar el codi del fitxer:

- El que hi ha entre els guions `---` és una capçalera (*header*), que conté títol, autor, data, i on es poden posar determinades opcions addicionals.
- Hi ha trossos (*chunks*) de codi **R** entre tres *backticks* (l'accent obert) `````.
- Hi ha títols d'apartats en línies que comencen amb `##`.
- La resta és text que surt tal qual en el HTML, però veiem que: L'adreça web que hi ha entre parèntesis angulars `< i >` s'ha tornat "activa"; la paraula emmarcada en doble estrelles `**` ha sortit en negreta; els caràcters entre backticks simples ``` han quedat formatats com els chunks.

Examinem ara el codi **R** entre triple backticks, començant pel segon:

```
1 ```{r cars}
2 summary(cars)
3 ```
```

A la primera línia hi ha, entre claus, la lletra `r`, que indica que el que segueix és codi **R**, i la paraula `cars`. Aquesta paraula és simplement una etiqueta per si més tard volem referir-nos a aquest *chunk*. La podríeu eliminar i no passaria res en aquest exemple.

Examinem ara l'últim chunk:

```
1 ```{r pressure, echo=FALSE}
2 plot(pressure)
3 ```
```

Després del l'etiqueta opcional `pressure`, i separades per comes, poden haver-hi algunes opcions, que s'apliquen només a aquest chunk. L'opció `echo=FALSE` fa que el codi no s'imprimeixi, però l'execució es fa i el resultat es presenta.

En el primer chunk

```
1 ```{r setup, include=FALSE}
2 knitr::opts_chunk$set(echo = TRUE)
3 ```
```

s'estableixen unes opcions que seran vàlides per tots el chunks, si no es diu el contrari (en aquest cas, l'opció és `echo=TRUE`).

L'opció `include=FALSE` d'aquest chunk particular indica que el **R** l'avaluarà, però no escriurà ni el codi ni el resultat. Això és útil per executar codi preliminar, però que no interessa explicitar en l'informe.

Una altra opció habitual és `eval=FALSE`. El codi no s'executarà, però sí que es mostrarà si `include=TRUE`, que és el valor per omissió.

**Exercici:** Canvieu `cars` per `mtcars`, que és un data frame que ja hem usat abans i té més dades. Afegiu al primer chunk els primers elements del data frame amb la funció `head()`, abans del `summary()`.

Canvieu el `plot(pressure)` per un histograma de la variable `mpg` i un núvol de punts de la variable `mpg` contra la variable `qsec`. Feu que el codi també surti al document.

Afegiu un altre apartat amb títol `Models estudiats` i feu que surtin els noms dels cotxes ordenats alfabèticament. (Necessitareu la funció `sort()`.)

**65 §** Aneu provant el que segueix escrivint qualsevol text en el nou apartat que heu creat en el fitxer `.Rmd`. Diferents paràgrafs se separen amb una línia en blanc; en general és aconsellable deixar línies en blanc entre diferents elements del text.

- El text que s'escriu entre blancs subratllats o estrelles (`_text_`, `*text*`) queda en lletra inclinada; el que s'escriu entre dobles estrella (`**text**`), queda en negreta.
- Podeu crear enllaços (hyperlinks) a pàgines web fent `[enllaç a la meva web preferida](http://cv.uab.cat)`.
- Es poden escriure fórmules, amb la mateixa sintaxi que veurem en  $\text{L}^{\text{T}}\text{E}^{\text{X}}$ . Proveu d'escriure aquest paràgraf:

```
La mitjana  $\bar{x}$  d'un conjunt de valors  $x_1, \dots, x_n$ 
es defineix com
$$
\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i
$$
```



Observeu que en el mateix panel on esteu editant es mostra una previsualització de la fórmula.

- Els títols de seccions, subseccions, etc, van precedits de #, ##, ###, fins a sis nivells, de més gran a més petit. Proveu de fer una subsecció de la secció que esteu fent.
- Llistes numerades comencen amb números, i llistes no numerades es poden començar amb -, \* o +. Es poden fer subllistes indentant:

Algunes mesures bàsiques:

1. la mitjana
2. la mediana
3. la variància
  - corregida ( $\frac{1}{n-1}$ )
  - sense corregir ( $\frac{1}{n}$ )

- Per fer taules una mica ben presentades, va bé la funció `kable()` del paquet `knitr`. Proveu això:

```
1 ```{r tables-mtcars}
2 knitr::kable(mtcars[1:5, ],
3             caption = 'Primeres files del data frame **mtcars**:')
4 ```
```

- També podem escriure codi **R** dins d'un paràgraf de text, usant *backticks* simples. Proveu:

La mitjana de mpg de tots aquests cotxes és ``r mean(mtcars$mpg)``  
 Això és millor que posar directament el valor numèric, perquè si ha algun canvi en la taula, el valor quedarà automàticament actualitzat.

**66 § (Opcional)** El mecanisme que hi ha darrera del que acabem de veure és el següent:

El `rmarkdown` utilitza dues eines: El paquet `knitr`, que el `rmarkdown` instal·la automàticament, i el `pandoc`, que és un programa extern que el `RStudio` ens ha instal·lat també automàticament.

La funció `knitr::knit()` llegeix el nostre fitxer `.Rmd`, executa els chunks de **R**, i produeix un fitxer `.md` en un llenguatge anomenat **Markdown**, que ja no conté codi **R**. Després, el `pandoc` transforma aquest fitxer **Markdown** en el codi HTML final.

$$\text{test.Rmd} \xrightarrow{\text{knit()}} \text{test.md} \xrightarrow{\text{pandoc}} \text{test.html}$$

Quan premem el botó `knit`, el `RStudio` executa la funció `rmarkdown::render()`, com veieu a la pestanya **Render**, i aquesta és la que desencadena el procés anterior.

Aquesta idea de dues etapes és molt potent, perquè el `knitr` és capaç de processar altres llenguatges de programació apart del **R** (per exemple Python), i altres llenguatges de procés de text (*markup languages*), apart del **Markdown** (per exemple  $\text{\LaTeX}$ ); i per altra banda el `pandoc` transforma fitxers entre una gran quantitat de formats (vegeu <https://pandoc.org/>). El podeu utilitzar independentment del **R** i el `RStudio`.

Un detall important és que el codi del fitxer `.Rmd` s'executa en una sessió nova de **R**, diferent de la que estàvem. Per tant, no hi ha cap objecte definit inicialment. D'aquesta manera, és segur que processant el `.Rmd` en moments o ordinadors diferents, el resultat final serà sempre el mateix.

**67 §** Tornem a mirar la capçalera del document (entre els guions `---`). Fixeu-vos en el ``r Sys.Date()``. Estem cridant a una funció de **R** que ens escriu la data d'avui. Si voleu deixar una data fixa, podeu substituir-ho per qualsevol text entre cometes. Per exemple, comenteu amb `#` aquesta línia i escriviu a sota `date:` amb la data d'avui.

Fixeu-vos ara en `output: html_document`. Això ens diu que prement el botó `knit` es crearà una pàgina web en HTML. Si posem `output: word_document` ens crearà un fitxer de Word. Observeu també que el botó `knit` és desplegable, i ens apareixen les opcions HTML, PDF, i Word. Proveu de produir un Word així, sense tocar el fitxer. L'opció PDF la podreu usar quan tingueu el  $\text{\LaTeX}$  instal·lat.

Hi ha també altres opcions de presentació del document que es posen en aquesta capçalera. Per exemple, feu que `output` tingui aquest aspecte (respectant el sagnat!) i mireu el resultat:

```
output:
  html_document:
    toc: TRUE
    toc_float: TRUE
    number_sections: TRUE
```

**68 §** Apart del `echo`, el `include` i el `eval` que hem vist, al principi de cada chunk es poden posar altres opcions. Per exemple, la mida i posició de les figures es pot controlar amb `fig.width=`, `fig.height=` i `fig.align=`. Els dos primers prenen una dimensió (en polzades), i l'últim una de les opcions `"default"`, `"left"`, `"right"`, `"center"`.

**Exercici:** Proveu-les sobre les figures que tenim al fitxer.

**69 §** Canvieu ara en la capçalera el `html_document` per `html_notebook` i guardeu el fitxer. Veureu que el botó `knit` canvia a `Preview`, però si el premeu el resultat és semblant.

La diferència és que podem fer que la sortida de les instruccions aparegui en el propi panel de l'editor, anàlogament als Jupyter notebooks de Sage. En el desplegable `Run` de la dreta, feu `Run All Chunks Below` i observeu el resultat.

Es poden executar els chunks individualment, però recordeu que per donar per bo un document, cal fer una execució completa i comprovar el que surt, exactament igual que amb el **Sage**.

**70 §** És possible també crear documents **RMarkdown** interactius. Anem a fer-ho amb un fitxer nou. Feu `File > New File > R Markdown...`, i marqueu `Shiny` i `Shiny Document`. Noteu que en la capçalera apareix `runtime: shiny`.

Guardau-lo amb un nom, com ara `testShiny.Rmd` i premeu `Run Document`. Ara podeu jugar amb el document com si estiguéssiu en una pàgina web interactiva.

Les funcions de **R** necessàries per produir aquests resultats (`inputPanel()`, `selectInput()`, `sliderInput()`, `renderPlot()`) pertanyen al paquet `shiny` i no l'estudiarem en aquest curs.

**71 §** Podeu descarregar-vos una *cheat sheet* força completa de **RMarkdown** des del mateix menú del **RStudio**, fent `Help > Cheat Sheets > R Markdown Cheat Sheet`.

Una altra més simple la trobareu a <https://www.rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf>.

# Índex

1	Instal·lació del <b>R</b> i el <b>RStudio</b>	1
2	Engegar el <b>RStudio</b>	2
3	Primeres passes	2
4	Help!	6
5	Objectes	7
6	Scripts	13
7	Més objectes: matrius i data frames	15
8	Gràfics i funcions	19
9	Paquets	22
10	Llegir i escriure fitxers amb dades	25
11	Estructures algorísmiques	29
12	Les funcions <code>apply()</code> i família	36
13	Entorns	37
14	Precisió en els càlculs	41
15	Rmarkdown	45